The role of the crucial experiment in student modelling

Thesis

The Role of the Crucial Experiment in Student Modelling

Rick Evertsz BSc

Thesis submitted for the degree of
Doctor of Philosophy in Artificial Intelligence, IET,
The Open University,
U.K.
September 1990

Abstract

As the range of models which tutoring systems can capture is extended, efficient diagnosis becomes more difficult. This thesis describes a solution to this problem based on the generation of 'Critical Problems'; their role in student modelling is analogous to that of the 'Crucial Experiment' in science. We argue that great diagnostic power can be obtained by generating discriminatory problem examples. In general, efficient diagnosis is just not possible without such an hypothesis-testing capability. We describe a program, PG, which given a pair of production rule models and a description of the class of problems which the student must solve, generates an abstract specification of the problems which discriminate between those two hypotheses. Through a process termed 'Abstract Interpretation', PG tips the balance in favour of diagnostic measurement. The key to this problem lies in the realisation that we are only interested in the abstract mapping between a model's inputs and outputs; from the point of view of generating a Critical Problem, the intermediate processing of the model is irrelevant.

Acknowledgements

Mark Elsom-Cook jointly supervised this research. Through regular weekly meetings, he provided the support and provocative guidance required to bring this research to a successful conclusion. The field of ITSs is by its very nature multi-disciplinary. Mark is one of the very few who have a full command of the many relevant areas, thus I was very fortunate to have him as a supervisor. During the many sticking points in this research, he was there with the help and advice needed to find a way forward. Once I had left the Open University he was always there in the background, gently prodding me to finish writing-up, and thereby ensuring that I did not lose momentum for too long.

During my first degree, Tim O'Shea was kind enough to allow me to do a placement at the Open University. This was my first experience of the role which AI can play in tutoring systems, and fuelled my ambition to learn more of this area. That golden opportunity was the starting point of the road which led to this thesis. During the years when I was registered part-time, he was my joint-supervisor and gave a lot of encouragement - always ensuring that I had the time needed to keep this research "on the boil". During what was a very trying period for him, he spared a great deal of valuable time to comment on the final draft. To me, that says it all about his kindness and generosity.

Marc Eisenstadt was my supervisor when I was based in the Human Cognition Research Laboratory at the Open University. For those first two years he provided lots of advice, and helped turn my half-baked ideas into a tractable research plan. In particular, he devoted a lot of time and effort ensuring that my fellow postgraduate students and I had the right computing facilities to carry through our research.

Tony Hasemer started this whole thing going, by encouraging me to register for a PhD. In the early days back in the RAF Hut, he taught me most of the Lisp I know. Through many hours of help and tutoring, he was responsible for turning me from someone who dabbled with Lisp into a member of that breed termed the "Lisp Hacker". Most hackers have a love/hate relationship with programming, me included, therefore I sometimes wonder whether I should thank him or blame him for that! Seriously though, in my more sane moments I realise that I owe him a very great debt.

Rachel Hewson offered lots of advice on all sorts of typographic matters which I had not even considered. Earlier drafts of this thesis reveal how much the layout has been improved by her input. Of course, she is not responsible for my inevitable failure to remain faithful to all of the principles she described.

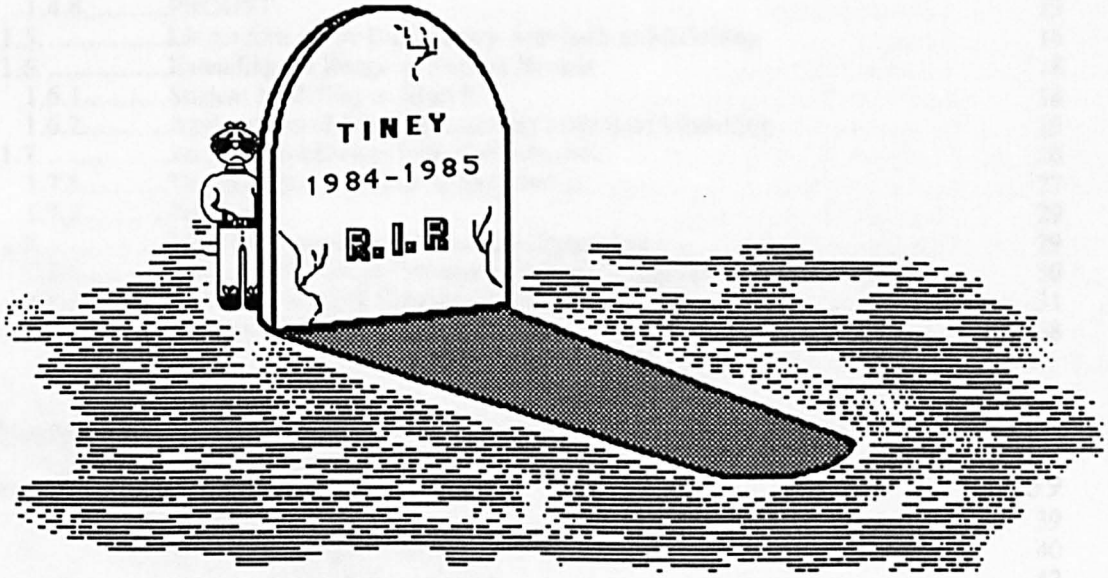
Many thanks to Mike Brayshaw who valiantly braved the formal detail of Chapter 3. His vetting of, and advice on, the Prolog definitions was invaluable.

I am also indebted to Mark Dalgarno, Roshni Devi, Pat Fung and Stuart Watt for checking earlier drafts of this thesis. They spotted some real howlers.

Dedication:1

To Jari, Asha and Natalia.

Dedication:2



To TINEY,

that tortuous route which I almost followed.

Table of Contents

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Table of Figures

Chapter 1

Chapter 2

Chapter 3

Chapter 4

No figures

Chapter 5

Chapter 1

Student Modelling and Problem Generation

1.1. Goals and Motivation

In the realm of Intelligent Computer-Aided Instruction (ICAI), the 'student model' is generally recognised as an essential prerequisite for adaptive tutoring. It embodies the machine's view of what the student 'knows' - without it, the tutoring system cannot make pertinent tutorial decisions. The process of modelling a student in the context of Intelligent Tutoring Systems (ITSs) is sometimes termed 'diagnosis'. Student modelling has much in common with other domains such as electronic fault diagnosis, and medical diagnosis: the overall goal of these systems can be described as one of obtaining an abstract account of some device's behaviour. For electronic fault diagnosis the device is a faulty circuit, for medical diagnosis it is the patient, and for ITSs, the student. Within this very general diagnostic goal, there are inter-domain differences in the type of diagnosis performed. Electronic circuit troubleshooters work from a *correct* model of the circuit, and try to determine why the faulty device's behaviour diverges from the ideal. Generally, ITSs try to produce a model of the student's behaviour which is not necessarily based on a *correct* model (those that do are said to be performing 'Perturbation Modelling'). This is analogous, in the circuit troubleshooting domain, to generating a model of the faulty device, without being given any information about the actual components and wiring of the circuit.

In these domains, the desired accuracy depends on the requirements of the diagnostician. With electronic fault diagnosis, the goal is to obtain *just enough* information to fix the fault; in medicine, the diagnostician is only concerned with curing the patient. Similarly, it is sufficient for a tutoring system's model of the student to be just accurate enough to guide its tutorial actions - further accuracy is superfluous. This view of modelling contrasts with that adopted in pure science, where one is (in an ideal world) searching for the most accurate model of the objects of interest. Thus, the problem of student modelling is more limited than that of cognitive modelling. Nevertheless, the space of models which a tutoring system must consider is non-trivial. Current student modelling research is concerned with extending this space through the use of machine learning techniques (cf. ACM (Langley & Ohlsson, 1984)). Whilst the machine learning approach does extend the potential coverage of the modelling system, the larger search space leads to poor runtime performance - as it stands this method could not form part of a tutoring system which runs in real-time. Interestingly, research on electronic fault diagnosis has run into similar problems. Extending the range of models to the multiple-fault case, leads to a mushrooming of an already large search space (DeKleer, 1986). Thus for diagnostic domains, a major research goal is to provide the machine with *efficient* methods of searching ever larger spaces.

Through the medium of a computer program called PG (Problem Generator[1]) the research described herein investigates an approach to controlling the diagnostic search space, whose roots lie in the methodology of science - where the experimental method is employed to distinguish between competing hypotheses. The central tenet is that great diagnostic power can be obtained by generating discriminating problem examples, to pose to the student. Armed with the ability to test hypotheses about what the student is doing, the modeller acquires the same powerful leverage available to the scientist who can generate experiments to test his/her theories. In general, efficient diagnosis is just not possible without interrogating the device under study. This is because the diagnostician would be forced to keep *all* consistent hypotheses open right up until a final choice is made. Many of these hypotheses could be discarded (or at least, devalued) early on by appealing to the device for relevant data (measuring values at judicious points in the circuit, or, in the case of student modelling, entering into a diagnostic dialogue with the student).

[1]PG was implemented in InterLisp-D on a Xerox 1109. It has now been ported to Common Lisp.

1.2. Overview

We have then as our main goal the desire to develop an automatic method of discriminating between the competing diagnostic hypotheses in a tutoring system. This problem reduces to one of discriminating between *two* competing hypotheses, for if we can solve this problem, then we will have cracked the multiple hypothesis case. This is because multiple hypotheses can be tested by pairwise comparisons amongst them. PG's approach to this problem is to analyse the computational behaviour of each model and then to derive an abstract description of how the two models differ. This description is then used to generate a discriminatory problem.

This chapter contains a review of previous approaches to student modelling, and a discussion of some of the limitations of such models. We then discuss some of the theoretical limitations of the notion of a discriminatory problem. The chapter concludes with an in-depth analysis of two previous approaches to problem generation, and concludes that though they both have something to offer, neither is sufficiently general.

Chapter 2 begins by describing how our problem relates to the general one of designing experiments to test scientific theories. It is concluded that experimental design is facilitated by a theory which makes the relationship between *observable* phenomena explicit; thus, we decide to develop a method of extracting the relationship between the observables of models expressed as production systems. We then describe the production rule language developed for this study. The latter half of the chapter is concerned with describing our first exploratory attempt to solve this problem. This approach was marginally successful but had some serious drawbacks which entailed its rejection.

Chapter 3 develops a new approach to the problem; the computational behaviour of the models is derived through a process of 'Abstract Interpretation'. The algorithm is presented in Prolog in an incremental fashion, first without and then with conflict resolution. We also discuss the theorem proving abilities of the program and how to derive an abstract problem from the input/output mappings of the model pairs.

The next chapter contains an evaluation of the problem generation abilities of our implementation. PG is found to cope well with models which wrong-footed the implementation presented in chapter 2. The main evaluation of PG is based on a set of fraction subtraction models derived from a separate study. Although PG successfully handles these models, it would not be able to cope with models which embody loops.

The final chapter presents some ideas on how to augment PG so that it can reason about loops. We discuss how one might integrate PG with a tutoring system and conclude with a discussion of other problems to which PG could be applied. Shorter descriptions of this work can be found in Evertsz (1989a; 1989b; 1990).

1.3. A Problem Generation Scenario

Imagine a situation in which a tutoring system is trying to ascertain how a student deals with fraction subtraction problems of the form: $W - X^y/_z$. The student has just solved the problem $8 - 4^3/_4$, and come up with an answer of $4^3/_4$. The tutor searches for an account of the student's error, and produces two possibilities:

> Model-1: If the first whole number is loose[1], then subtract the second whole number from the first, and write down the fractional part of the second term.
>
> Model-2: If the first whole number is loose, then the answer is the whole of the second term (i.e. discard the first term).

The tutoring system tries to obtain more information by setting the student a problem of a similar type (i.e. of the form $W - X^y/_z$). Instantiating this problem template with a random set of numbers (subject to the constraint that it form a legal fraction subtraction problem), the tutor sets the following problem next: $4 - 2^7/_8$. Unfortunately, in the current situation this problem provides no diagnostic power whatsoever, as shown below:

[1] A 'loose' whole number is one without an associated fraction, e.g. 4 rather than $4^1/_4$.

Processing for Model-1: $4 - 2^7/_8$
$4 - 2 = 2$
copy $^7/_8$
answer is $2^7/_8$

Processing for Model-2: $4 - 2^7/_8$
copy $2^7/_8$
answer is $2^7/_8$

In order to discriminate between the above two models, the machine must generate a problem which will produce a different answer for each (more specifically, each model should produce different machine-analysable output behaviour; it actually doesn't matter whether the answers are the same, as long as some intermediate model-distinguishing behaviour is available to the machine). This entails deriving an abstract description of the transformations performed by the production rules in each model. For the two models above, the abstract answer descriptions would look as follows (**x**, **y**, and **z** are variables; expressions in italics must be computed):

Model-1
For an input of the form: WholeNumber1(x),
WholeNumber2(y),
Fraction2(z),
the output is of the form: Answer(*subtract(x,y)*,z)

Model-2
For an input of the form: WholeNumber1(x),
WholeNumber2(y),
Fraction2(z),
the output is of the form: Answer(y,z)

A tutoring system, armed with these abstract descriptions, is in a position to generate a problem which discriminates between them (termed a Critical Problem or 'CP'). In the above example, the difference between them lies in the first term of the answer. For model-1 it is the result of evaluating the expression ***subtract(x,y)***, whilst for model-2 it is just the variable y. Thus, the problem generator's task is to find an **x** and **y** such that ***subtract(x,y) is not equal to y***. Values of '3' for **x** and '1' for **y** will do nicely (the value of **z** is irrelevant), giving the problem: $\mathbf{3 - 1^7/_8}$.

The above example illustrates why tutoring systems which model the student, should be able to test their diagnostic hypotheses. However, real student models are far more complex than the trivial ones above. If cast as a production system, student models typically consist of many rules, and may involve such computational behaviours as conflict resolution, iteration, and complex pattern matching. In order to generate CPs for such models, the program must be capable of *reasoning* about what the models compute. Only then can it compare two models to determine under what conditions their behaviour differs.

In order to set this work within the framework of student modelling, we now review previous work in this area. This is followed by a critique of earlier attempts at CP generation.

1.4. An Overview of Student Modelling

The student model is central to the notion of adaptive teaching. A system's ability to adapt to the needs of a student is wholly dependent on its perception of what those needs are. By keeping an accurate record of the student's knowledge, the tutoring system is better able to assess the student's needs.

Both the knowledge encoded, and the representation of that knowledge can vary a great deal. This section presents a brief overview of these various approaches to student modelling, and thereby outlines the research background from which PG stems.

1.4.1. SCHOLAR

SCHOLAR (Carbonell, 1970) is a tutoring system whose domain of expertise is South American geography. The domain knowledge is represented as a semantic network of geographical concepts and facts. This network is also used to model the student. SCHOLAR maintains the student model by annotating the nodes and links to record their mastery by the student (i.e. they are 'ticked off' when the student answers the associated questions correctly). This technique is now termed 'overlay modelling'. In the 'overlay' method, the system's domain expertise is cast as a collection of 'knowledge units', and the student's learning is

characterised as a subset of those units. This approach is also referred to as 'subset modelling'. Naturally, this method can only model those aspects of the student's behaviour which are 'correct', as defined by the tutor's conceptualisation of the domain. One can extend this approach to include 'incorrect' pieces of knowledge, yielding a student model which is an overlay of the union of the correct and incorrect model components ('perturbation modelling').

Although SCHOLAR's student model is very simple, it was an important early attempt to incorporate a model of the student, expressed in terms of the domain being tutored.

1.4.2. WEST

WEST (Burton and Brown, 1982) is a computer-based game where the goal is to get from the start to the home town. On each turn, the machine generates three random numbers, which the player must compose into an arithmetical expression consisting of the three operands only (no duplicates). The value of this expression determines how far the player can move. The game allows for shortcuts and 'bumping' the opponent's piece backwards; thus, the student must consider various strategies - it is not sufficient to find the maximum value for the expression, smaller values may allow the player to take a shortcut.

Burton and Brown characterise the game in terms of 'issues' and 'examples'. The term 'issues' refers to the concepts and skills which the student is expected to acquire. Each issue consists of a 'recogniser' and an 'evaluator'. The recogniser is used to spot instances of an issue's use, whilst the evaluator is used to determine whether the student is weak in that issue. When the student makes a move, WEST runs the issue recognisers to identify the issues used by the student, and compares these with the issues which it would have chosen itself. WEST derives a 'differential model' of the student's behaviour by comparing the two sets of issues to determine where the student is weak. If the tutor decides to interrupt, it selects an issue which is better than the one used by the student, and instantiates it to produce an actual move ('example') which the student could have used.

WEST's differential model keeps a tally of how often the various issues were used appropriately, and how often they were missed (as defined by WEST's preferred choice of issue in each case). It includes information about the abstract expressions used/missed by the

student, special moves used/missed (e.g. 'bumps'), and strategies used (e.g. how many times the student chose maximum distance).

1.4.3. WUSOR

WUSOR-I (Stansfield, Carr and Goldstein, 1976) is designed to coach a player in the adventure game WUMPUS. The player moves from cave to cave in search of the Wumpus, encountering hazards on the way, and, if successful, wins by firing an arrow into the Wumpus' lair. WUSOR-I did not possess a student model and so was quite limited. WUSOR-II incorporates an overlay model of the student. The student model is represented as a subset of the tutor's 'expert' set of rules for playing the game. Individual rules within the subset are annotated by a number representing the proportion of times the rule was appropriately applied (i.e. applied when the tutor would have made the same choice).

Goldstein (1982) proposed a formalism called a 'genetic graph' as an extension to WUSOR-II. In this scheme, the procedural rules form the nodes of the graph, interlinked by relations such as generalisation/specialisation, analogy, deviation/correction, and simplification/refinement. These genetic links specify the evolutionary relationships between rules. Goldstein argues that the extra information provided in these links, enables more refined modelling of the student than that provided by plain subset modelling. For example, the deviation links allow one to represent faulty knowledge explicitly. Furthermore, viewed as a partial learning theory, the links enable the tutor to reason about what skills the student is ready to acquire.

1.4.4. The BUGGY Model

The seminal work on diagnosing bugs in procedural skills was that of Brown and Burton (1978). A common misconception amongst workers in education was that errors in procedural skills, such as long division and multi-column subtraction, were the result of such factors as careless setting-out of exercises, lack of concentration, or an inability to follow the correct procedure (e.g. Downes & Paling, 1958). Disputing the notion that the error-prone student has difficulty following the 'correct' steps of an algorithm, some workers in education have

posited that students are actually consistently following an erroneous algorithm (Ashlock, 1976). With this objection in mind, Brown & Burton developed the 'Buggy model' of errors. They had observed that students are remarkably good procedure followers, but are often executing an erroneous procedure - one which contains discrete modifications to the correct skill. The erratic behaviour was only manifestly so, being the result of one or more deep, systematic bugs.

To model the different algorithms, Brown & Burton chose to use a procedural network representation. Essentially, their model represents arithmetical skill as a hierarchically organised collection of subroutines, where each subroutine represents a particular subskill. Bugged behaviour is simulated by replacing a correct subroutine with a faulty one. From a psychological point of view, this representation is unsatisfactory. The problem is that one cannot predict in what way different parts of a skill can become bugged, because these predictions would be based on manipulations occurring at the implementation level of the model (Young & O'Shea, 1981). To the extent that their model divides the skill into discrete subskills, it can be used to predict which portions of the skill are open to corruption, but it cannot predict the types of bug affecting each subskill. Brown & Burton accept this criticism, saying that they chose an ad hoc representation because they did not know in advance what primitives or control structures would be appropriate for modelling simple procedural skills (Burton, 1982).

The buggy model of subtraction has been incorporated in three programs: BUGGY, DEBUGGY, and IDEBUGGY. BUGGY was used to train teachers in the art of bug diagnosis. DEBUGGY and IDEBUGGY are diagnostic systems, the former works on the results of tests taken by students, whilst the latter diagnoses students interactively. In section 1.5 we will evaluate the performance of the diagnostic system DEBUGGY, and in section 1.8.1 we will examine IDEBUGGY's problem generation capabilities.

1.4.5. Production Rule Models as a Representation for Student Models

Efforts to model the flexible control structure of human problem solving behaviour, led to the use of Production Systems (PSs), (Newell & Simon, 1972). Since then, PSs have become the psychologist's favoured representation for modelling problem solving. With PSs, we have the opportunity to model the very flexible control structure of human behaviour. The

information which triggers the rules is held in a single database - only this data can have control over behaviour. This makes PSs highly distractable: if some important event occurs (represented as an item in Working Memory), then, assuming that Production Memory contains a rule which deals with such an event, the system reacts immediately. This contrasts favourably with representations, such as DEBUGGY's procedural network, where the program 'cranks' through to the end, oblivious to any potentially useful features of the problem state. For example, if one were to suddenly insert a problem's answer into Working Memory, then a PS in the middle of solving that problem would take advantage of this unexpected bonus, and stop (unlike DEBUGGY).

The other advantages of PSs stems from their modularity. Because rules do not explicitly invoke each other, but instead must communicate via Working Memory, PSs can withstand small perturbations to their rule-set. Researchers have found this attribute useful in modelling errors which can be characterised as deletions to the correct model of some procedural skill (e.g. Young and O'Shea, 1981). To model such errors, one simply deletes the appropriate production rule from the rule-set. Provided that there is sufficient redundancy in the rule-set, such a PS will still come-up with an answer, albeit the wrong one. Modularity also eases the problems of modelling learning, because it is relatively easy to extend the system by adding discrete components of the skill to the model. From a production rule perspective, learning a procedural skill entails adding new productions to Production Memory.

With these advantages in mind, PSs have been employed by Young and O'Shea (1981) to model children's subtraction, by Sleeman (1981) to model students' algebra, and by Evertsz (1982) to model children's fraction subtraction. Attempting to provide a principled account of the kinds of errors found by Brown & Burton, Young and O'Shea modelled Bennett's (1976) subtraction data using PSs, and found that they could account for a large portion of the observed errors in children's subtraction, by the deletion of one or more rules from the 'correct' model of the skill. For example, deletion of the 'borrowing rule' allows the PS to carry-on, but produces the wrong answer for problems where a borrow is required. Unfortunately, the 'missing rule' account is not as principled as one would like; Young and O'Shea also had to include mal-rules to account for the errors. These mal-rules were 'principled' in the sense that they were derived from rules appropriate to other arithmetic skills. These rules were all concerned with coping with zeros. For example, some children, when solving a multi-column subtraction problem, would write down the number 'N' for columns of the form '0 - N'. This could be the result of generalising the rule: '0 + N = N'

from addition. This notion of erroneously generalising a rule from a related task is a compelling one. Matz (1982) presents an interesting model of errors in algebra, founded on a similar hypothesis, and describes the particular extrapolation techniques which are used to modify a rule so that it can be applied to the new problem. Young and O'Shea's model accounts for about two thirds of the non-number-fact errors by the deletion of rules, and the inclusion of 'zero-pattern' errors, as described above. In addition, their model accounts for Brown and Burton's 15 'most frequent' subtraction errors.

In a similar study of the domain of fraction subtraction, Evertsz (1982) found that 68% of the errors were systematic. However, to achieve this coverage, it was necessary to include mal-rules which were not taken from other arithmetic skills. Furthermore, each child's production system (generated from half of the problems) predicted, on average, 79% of the child's (non-number-fact) errors (tested using the other half of the problem-set). Unlike the earlier subtraction studies, the inclusion of mal-rules was crucial to getting adequate coverage (71% of the systematic errors were of the mal-rule variety). Thus, it would seem that the domain of fraction subtraction requires children to be a little more 'inventive' in trying to get an answer. It is interesting to note that, in keeping with Matz's view of bug development, half of the mal-rules could be viewed (informally) as mutations of the operators and operands of rules from other arithmetic skills.

From the student modelling perspective, the promise of Young and O'Shea's work lay in the prospect of being able to predict the range of empirically-observed errors, through simple mutations of a correct model of the skill being taught. However, this promise has not been fulfilled; the approach does not extend to other domains such as fraction subtraction (Evertsz, 1982) and algebra (Sleeman, 1981).

1.4.6. LMS/PIXIE

LMS (Sleeman and Smith, 1981), later renamed PIXIE, is a student modelling system developed for the domain of algebra, in which the models are expressed as ordered production rules. Unlike Young and O'Shea, Sleeman and Smith do not model errors as missing rules, but rather as modified versions of correct rules. There are no constraints on these deviant rules (mal-rules) other than that they must have the same lefthand side as their progenitor. Thus,

like BUGGY the mal-rules (buggy procedures) are syntactic variants of the 'correct algorithm', with no underlying theory constraining their form.

In order to contain the combinatorics of the modelling task, LMS models algebra subskills first and then gradually extends the coverage of the model, as the student answers more complex problems. This approach was later found to be flawed, because students who have mastered a given subskill on its own, cannot always apply it when it is part of a more complex problem. Later, Sleeman began augmenting LMS with a problem generator; this is reviewed in section 1.8.2.

1.4.7. Repair Theory

Repair Theory (Brown & VanLehn, 1980) is a generative theory of bugs in procedural skills. In other words, it is an attempt to predict the range of buggy procedures acquired by students. Developed using subtraction data, the authors hoped that the principles gleaned would be generally applicable to all procedural skills, so that one could create buggy models for completely new domains, without having to collect masses of empirical data first. The central tenet of Repair Theory is that bugs are the result of the student's attempt to 'repair' a missing part of their algorithm. The student, with a missing piece of knowledge, will eventually hit a problem requiring the application of that knowledge. At this point, the student is stuck; this is termed an 'impasse'. If sufficiently motivated, the student will try to carry-on. To this end s/he institutes a patch (termed a 'repair') across the missing piece of knowledge. This is expressed in the theory by the generation of a buggy-rule. The set of incomplete procedures is defined by applying a set of 'deletion principles' to the correct procedure. The repairs are defined by a set of 'repair heuristics', which propose repairs to a given impasse; this collection of repairs is filtered by a set of 'critics' which reject improbable repairs on the basis of the form of the solution which would (immediately) result. Thus, the set of possible bugs is the product of all possible impasses and all valid repairs.

Repair Theory was a laudable attempt to provide a principled account of bug acquisition. One important criterion, adopted by Brown & VanLehn, is that the theory should not predict what they term 'star-bugs', i.e. bugs which expert diagnosticians agree will never be observed in students. Consequently, the theory embodies a number of limiting constraints; for example the problem solver (which generates and tests repairs) is local in scope. It cannot look ahead to

see the effects of the proposed repair. Furthermore, the theory is very 'syntactic' in nature, and totally ignores the semantics of the procedures being executed.

These constraints limit the range of predicted bugs to 18 out of the 77 known at the time of the report (Brown & VanLehn, 1980). Given this low coverage, it seems likely that there are some bugs which may well always fall outside the realm of Repair Theory.

1.4.8. PROUST

PROUST (Johnson and Soloway, 1985) is a debugger for the programming language Pascal. In diagnosing the errors in a student's program, PROUST reasons about the *intentions* of the programmer. Programming is viewed as a three stage process in which the programmer first formulates his/her *goals* for the task, then converts them into programming *plans*, and finally instantiates the plans to produce actual program *code*. By reasoning about the student's programming goals, PROUST is able to diagnose errors which would be meaningless if viewed from outside the overall context of the student's intentions.

PROUST must be provided with a formal description of the programming problem being solved by the student. This information, coupled with the student's solution, is used to reconstruct the process by which the student derived the final program from the original problem specification. The reconstruction process begins with the expansion of the goals associated with the problem specification. Each of the goal's plans are matched with the parse tree of the student's program, and the one with the best fit is selected. PROUST can rationalise mismatches between a plan's predictions and the student's code by applying buggy rules which specify how faulty code can be generated from the plan.

PROUST's 'analysis by synthesis' strategy makes it quite different from the approaches discussed so far. This reflects the greater complexity of the programming process. Unlike multi-column subtraction, it cannot be adequately modelled as a hard-wired, procedural skill - the range of possible programming problems is just too huge to be amenable to a simple set of fixed procedures.

1.5. Limitations of the Bug Library Approach to Modelling

As we saw in the previous section, student models come in many shapes and sizes. In principle, they can range from a simple pass/fail pattern for the previous problems attempted by the student, to a cognitive model based on empirical studies of the nature of the skill being taught. In this thesis, we shall be concentrating on models of procedural skills only. Procedural skills are those where the student has to learn some relatively fixed procedure, such as algebraic manipulation, calculus, or multi-column subtraction. The student must learn the set of operators, as well as when it is appropriate to apply these operators to a given problem state.

To date, there are quite a few models of various procedural skills. One of these, DEBUGGY, was reviewed in section 1.4.4. In this section we examine DEBUGGY's coverage of observed errors (only a rough evaluation is possible).

It is rather difficult to evaluate the performance of bug-diagnosis programs (for a discussion of model-evaluation methods, see Priest and Young, 1988). The difficulty lies in deciding which evaluation metric is the best. There are many ways of evaluating a diagnostic program's performance, including:

(i) a measure of the overall percentage of problem answers predicted;

(ii) the percentage of subjects falling into each of a number of categories, such as
- correct algorithm,
- correct algorithm with one or two 'random' slips,
- consistently buggy algorithm,
- buggy algorithm with one or two slips,
- buggy algorithm with bug migration across problems,
- undiagnosed;

(iii) a measure of the degree of correspondence between the program's diagnoses and those of expert human diagnosticians.

It is not clear that any of these is the 'best' measure of performance. A measure of problem answers predicted overall is useful, but impoverished. All it really tells us is how consistent the students are in the program's eyes. It confounds student consistency with the program's diagnostic power. To illustrate, what does it mean to say that DEBUGGY correctly predicted 62% of the problem answers? It can mean one of at least two things, either: (a) the students are 100% consistent, and DEBUGGY is 62% successful in spotting their consistent algorithms, or (b) DEBUGGY is 100% successful at spotting systematic algorithms, but students are only 62% consistent, as they suffer from 'slips', or indulge in 'algorithm migration'. Classifying students, as in (ii), provides information on the cases where DEBUGGY failed to make the right prediction. However, there are worrying problems of interpretation in allocating subjects to each group. For example, it is not always clear whether the perceived inconsistency in a subject's answers is the result of a 'slip', 'bug migration', or some as yet undiscovered systematic algorithm.

In trying to evaluate the performance of DEBUGGY, we are fortunate in having VanLehn's retrospective analysis of DEBUGGY's success with multi-column subtraction (VanLehn, 1982). Comparing the performance of DEBUGGY with that of expert human diagnosticians, VanLehn concludes that the experts agreed more with DEBUGGY than they did with each other. The experts agreed with DEBUGGY's diagnoses in 83% of cases. Experts found buggy diagnoses for 13% of the students that DEBUGGY had classified as undiagnosed. This was due to the fact that the human experts had access to the students' whole solutions (including intermediate solution steps). DEBUGGY only had the students' final answers to work from. This extra information also led the experts to dispute 20% of the instances labelled as systematically buggy by DEBUGGY. DEBUGGY's ability to do so well, despite only working from the answers, is certainly impressive. Nevertheless, it is clear that it would have benefited from access to the students' intermediate jottings. Where a modelling system does not have the advantage of DEBUGGY's large bug-library for analysing buggy answers, access to student protocols is doubly crucial.

In the original study of some 1300 Nicaraguan school children, DEBUGGY classified the children's performance as 39% Buggy, 9% Slips, and 52% Undiagnosed. In a subsequent study of some 900 children the performance had 'improved' to 40% Buggy, 22% Slips, and 37% Undiagnosed. The improvement was partly due to new bugs added to the library in the interim, and partly due to the use of a highly diagnostic test in the second study, developed with DEBUGGY's help. This test enabled the program to discriminate more effectively

amongst the various possible interpretations of the results thereby reducing the number of anomalous (undiagnosed) answers.

But what of the 37% who foxed DEBUGGY? Is this a sad reflection on DEBUGGY, or is it the case that a lot of students will always escape diagnosis, because their problem solving behaviour is just too inconsistent? Let us return to VanLehn's analysis of DEBUGGY's unaccounted for 37%. VanLehn was able to make use of test-retest data to estimate how many of the errors were slips (slips are defined to be errors which do not appear on the same problem in both tests). Forty-five percent of the 37% were thus classified as having the correct algorithm, marred by several slips. A further 14% had slips interspersed with a buggy algorithm. Eighteen percent of the undiagnosed students were said to be 'tinkering'. This endearing little term refers to students who, during the course of the test, try several different repairs to one particular impasse. This still leaves 23% of the undiagnosed students unaccounted for, even by expert diagnosticians with the backing of 'scratch marks', Repair Theory, and test-retest data.

VanLehn (1982) states that: "The central limitation on DEBUGGY is its inability to invent new bugs." (pg37). Sifting through the scripts of the remaining 37% of students, for new bugs, is a non-trivial task. VanLehn estimates that, over the years, the six diagnosticians have put in some four or five thousand hours on hand diagnosis. The crucial question is, how many more hours must be spent before discovering all of the possible bugs? If we look at the approximate rate of acquisition of new bugs, we can see that it is tailing-off a little (figures 1-1, 1-2). It would appear that the bug-library is converging; but we have no way of estimating how much longer it would take for the discoveries to peter out. In an empirical study of elementary errors in algebra, Payne and Squibb (1988) conclude that bugs are unstable and that a large number of mal-rules are needed to encode them; furthermore, the mal-rules have little generality across schools.

Approximate time	bugs added
December 1977	45
October 1979	15
November 1979	30
April 1980	15
June 1980	10

Figure 1-1 — Rate of new bug discovery

(taken from VanLehn, 1982, pg38)

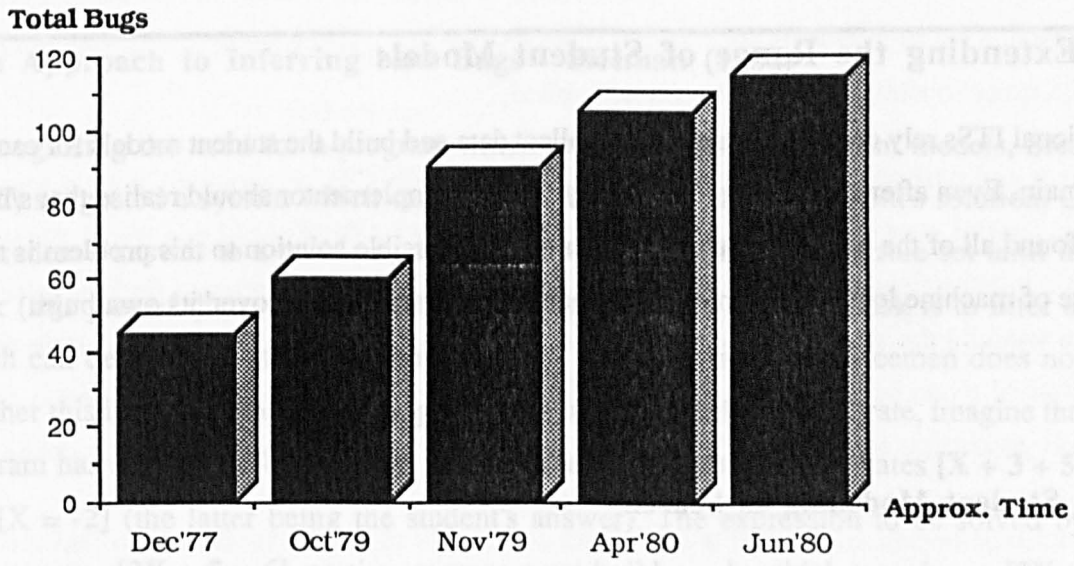


Figure 1-2 — Rough estimate of bug discoveries for DEBUGGY

As we shall see in the next section, one promising solution to this problem is to have the diagnostic program discover the new bugs for itself.

Summary

DEBUGGY does not provide a complete account of the errors made by students. The rate of discovery of new bugs in the domain of subtraction alone, leads one to suspect that there are

many more systematic bugs to be found. Repair Theory and the work of Matz, provide us with an idea of what bugs to expect in new domains. However, there are still bugs which fall outside these analyses. Ideally, we would like our tutoring system to cope with students who use novel algorithms. The approach adopted by ACM (Langley and Ohlsson, 1984) is a promising route to attaining this goal.

1.6. Extending the Range of Student Models

Traditional ITSs rely on the implementer to collect data and build the student models for each new domain. Even after completing this lengthy task, the implementor should realise that s/he has not found all of the bugs exhibited by students. One possible solution to this problem is to make use of machine learning techniques, so that the program can 'discover' its own bugs.

1.6.1. Student Modelling as Search

In the past, AI researchers have found it useful to recast many problems as 'search problems'. For example, solving the Tower of Hanoi problem can be viewed as 'search'. The program must search through the possible states in the problem space until it finds a route to the goal; this route is the solution path for the problem. Of course, merely saying that a task can be viewed as 'search' doesn't solve the problem. Many real problems have huge search spaces, thus the program must find a way of constraining its search through the space - searching the whole space is just not feasible. Learning can also be viewed as a search problem. Adopting this perspective clarifies many of the important issues, and helps one to characterise some of the differences between various learning programs (Mitchell (1982) has used this to good effect in analysing earlier learning programs, and in developing his Version Space algorithm).

The problem faced by the student modeller (in particular, one which uses data-driven machine learning techniques) can also be viewed as one of search. At one level, the program's goal is to find the route from the initial problem (start state) to the student's answer (goal state), drawing on its armoury of operators to move from state to state. In order to increase the range of models covered, one must extend this search space. The machine learning approach to student modelling is an attempt to do just that.

1.6.2. Applications of Machine Learning to Student Modelling

An Approach to Inferring New Bugs - Sleeman (1982)

Recognising the need for a program which can create its own student models, Sleeman (1982) suggested a system which can infer the 'missing' step in a student's solution. Given the student's answer to a problem, the system backward-chains on its rule-set until it gets stuck (this part of the system appears to be implemented). The next task is to infer a rule which can deduce the 'sticking-point' from the problem statement (Sleeman does not say whether this inductive portion of the program is implemented). To illustrate, imagine that the program has accounted for the steps between the (assumed) problem states **[X + 3 + 5 = 6]** and **[X = -2]** (the latter being the student's answer). The expression to be solved by the student was: **[3X + 5 = 6]**, so the program must build a rule which transforms **[3X + 5 = 6]** into **[X + 3 + 5 = 6]**. It is *possible* (though not necessarily warranted) to infer that the student's mal-rule is: **{MX+N=P => X+M+N=P}**; however, as Sleeman points-out, if we follow Neves (1981), then we should build a rule which embodies the essential differences between the two problem states: **{MX => M+X}**; in effect, we are generating as general a rule as possible.

Sleeman does not address the problem of modifying erroneously inferred rules. The rule, inferred above, may well be over-general; **{3X => 3+X}**, and **{MX & (CurrentState is startstate) => M+X}**, are equally valid (where the latter rule says convert **MX to M+X** only if you are working on the initial problem statement). Therefore it seems unlikely that the proposed generalisation method is up to the task of inferring all of the mal-rules which could be derived from the initial set of rules.

ACM

Perceiving the need for a less restricted form of student modelling, Langley and Ohlsson (1984) developed ACM. ACM is designed to circumvent the limitations of more traditional systems by using machine learning techniques to synthesise the algorithm used by the student. To tackle this problem, Langley and Ohlsson followed Newell (1980) in viewing human cognition as a search through some 'problem space', and adopted a production system formalism to effect that search. From this perspective, one must first define the problem space (or spaces) searched by the student. This entails defining a representation for the states in the search space, a set of operators for moving from state to state, and a set of conditions which define the states to which an operator can be applied.

Before it can commence modelling a student, ACM needs to be provided with a set of problems (initial states), and the student's set of answers to those problems (final states). Using this information, and its definition of the problem space, ACM tries to find the actual operators used by the student (by exhaustive search), and the conditions which constrain the application of those operators. A discrimination learning mechanism is used to derive the conditions governing an operator's use, based on viewing steps which lie off of the shortest solution path, as negative instances of the use of an operator.

To see the system in action, let's look at an example of ACM learning to perform multi-column subtraction, taken from Langley, Ohlsson and Sage (1984). Here, ACM learns how to subtract by working on the problem set: **{54 - 23 = 31, 93 - 25 = 68}**.

find-difference
If you are processing *column1*,
    and *number1* is in *column1* and *row1*,
    and *number2* is in *column1* and *row2*,
then find the difference between *number1* and *number2*,
    and write this difference as the result for *column1*.

add-ten
If you are processing *column1*,
    and *number1* is in *column1* and *row1*,
    and *number2* is in *column1* and *row2*,
    and *row1* is above *row2*,
then add ten to *number1*.

decrement
If you are processing *column1*,
    and *number1* is in *column1* and *row1*,
    and *number2* is in *column1* and *row2*,
    and *row1* is above *row2*,
    and *column2* is left of *column1*,
    and *number3* is in *column2* and *row1*,
then decrement *number3* by one.

shift-column
If you are processing *column1*,
    and you have a result for *column1*,
    and *column2* is left of *column1*,
then process *column2*.

Figure 1-3 — The initial production system for subtraction

(Langley, Ohlsson and Sage, 1984, pg12)

The initial operators are shown in figure 1-3. The two search trees are shown in figures 1-4 and 1-5. Block-shaped nodes represent those which lie on the solution path; circular nodes are negative instances. In these runs, ACM was presented with ten abstract condition types, of which seven are actually used for these two problems. ACM draws on these ten to find a subset which covers all of the positive instances, but none of the negative ones.

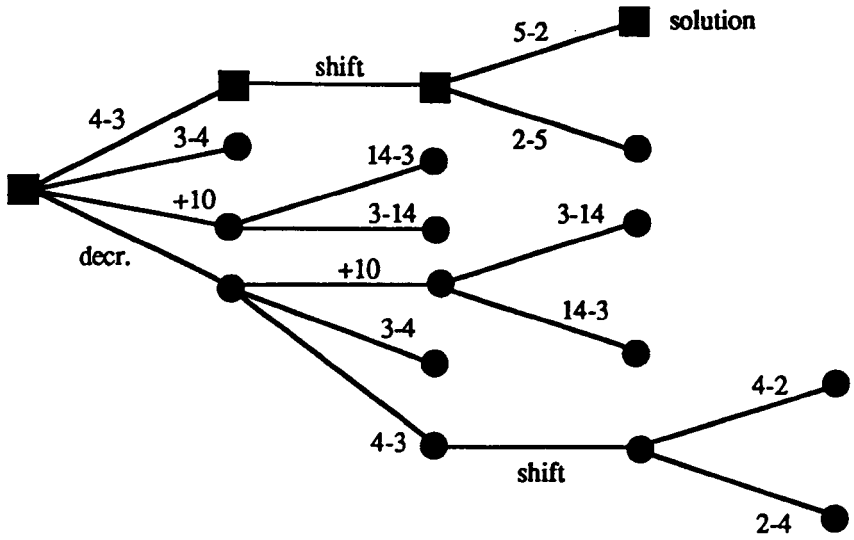


Figure 1-4 — Search tree for the problem 54-23=31

(Langley, Ohlsson and Sage, 1984, pg13)

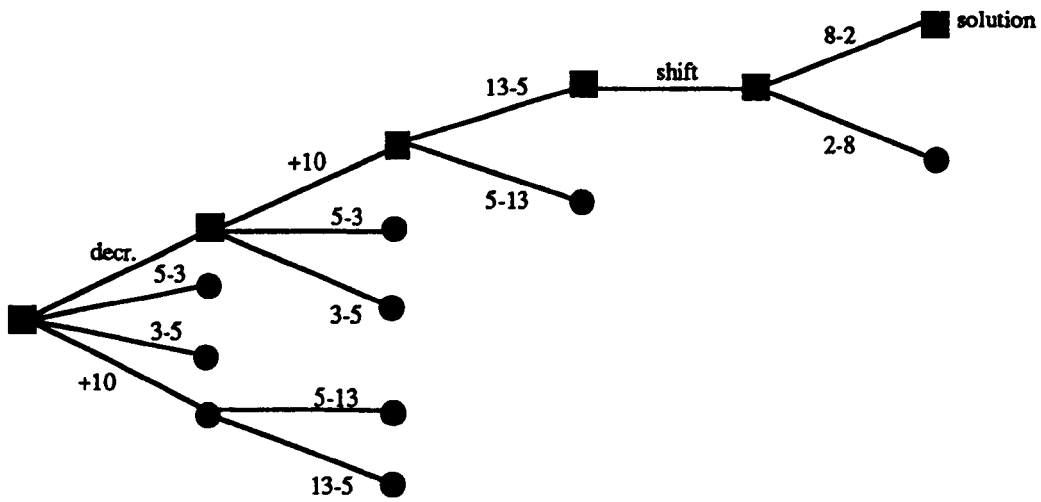


Figure 1-5 — Search tree for the problem 93-25=68

(Langley, Ohlsson and Sage, 1984, pg16)

The table, in figure 1-6, plots each condition against each operator instance from the two search trees. Cells containing an **X** signify that the condition is satisfied for that particular operator instance. For example, the condition **(greater n1 n2)** is satisfied by the instances: **13 - 5, 8 - 2, 4 - 3, 5 - 2, 5 - 3,** and **5 - 3'**. Instances below a '+' are positive, while

those beneath '-' are negative. Thus, the **greater** condition made an incorrect prediction in two situations. Similarly, the **above** relation was erroneously satisfied by two instances (**3 - 5**, and **3 - 5'**). ACM uses an evaluation function to order the tests to be added to the discrimination network[1].

	+ 13-5	+ 8-2	+ 4-3	+ 5-2	- 5-3	- 3-5	- 3-4	- 2-5	- 5-3'	- 3-5'	- 5-13	- 2-8	E
GREATER N1 N2	X	X	X	X	X				X				1.75
ABOVE R1 R2	X	X	X	X		X				X			1.75
ADD-TEN C1	X												1.25
DECR. C1		X											1.25
ADD-TEN-ANY	X	X									X	X	1.25
DECR-ANY	X	X							X	X	X	X	1.0
JUST-DECR. C1									X	X			1.25

Figure 1-6 — Tests and instances for the find-difference operator (Langley, Ohlsson and Sage, 1984, p17)

The table includes the evaluation score (E) for each condition. The conditions **greater** and **above** tie for the maximum score (in the event of such ties, the user must specify the winner). If we assume that the **greater** test takes precedence, then the discrimination net is initialised as shown in figure 1-7. The right branch contains only negative instances, while the left holds four positive and two negative.

[1]The evaluation function, E, is computed as follows:
E = maximum(S, 2 - S), where S is $M_+/T_+ + U_-/T_-$.
M_+ is the number of positive instances matching a given test.
U_- is the number of negative instances failing to match that test.
T_+ and T_- are the total number of positive and negative instances respectively.
Thus, the optimal score is 2 (the test matches all of the positive instances only, or all of the negative instances only), while the minimal score is 1 (matches half of the positive and half of the negative instances).

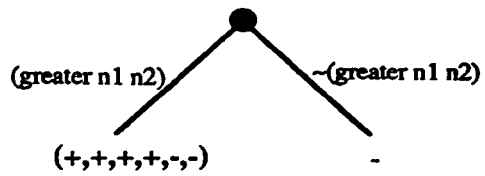


Figure 1-7 — Initial discrimination network

ACM now works out the evaluation function for the six instances remaining in the left branch. The **above** relation scores 2 (the maximum possible), giving the network shown in figure 1-8. The net building is now complete - all of the instances are discriminable on the basis of just two tests. The two conditions are added to the initial **find-difference** operator, ensuring that in future, when presented with these two problems, ACM will find the solution without search.

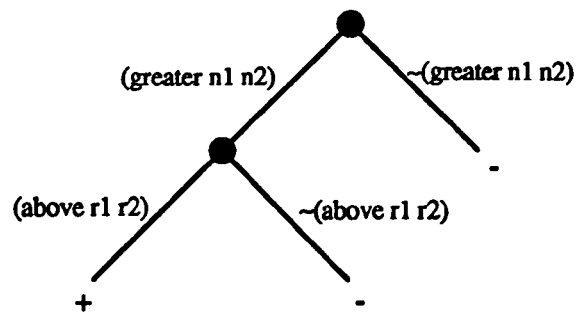


Figure 1-8 — Correct discrimination network for the find-difference operator (Langley, Ohlsson and Sage, 1984, p17)

Naturally, ACM can learn buggy procedures using the same method. Implemented on a Vax 750, ACM successfully modelled eleven common subtraction bugs, using idealised answers for a set of 20 test problems. To obtain this coverage, ACM had to be provided with some extra operators. Although this increased the search space considerably, ACM was still able to model the behaviour using exhaustive search, taking on average 2 CPU hours to generate each cognitive model on the basis of 20 problems. Clearly, much work needs to be done before this approach can form part of a real tutoring system.

ACM promises to extend the space of student models, but by how much? To represent a true advance over the bug library approach to modelling, it will need to show that it can discover novel (i.e. previously undocumented) bugs. It remains to be seen whether its small set of initial operators is sufficiently fine grained to cover algorithms which were unknown to, say, DEBUGGY. Whilst ACM is certainly more data-driven than its predecessors, there is still an important sense in which it is theory-driven. In providing the program with an initial set of operators, the user is relying on an implicit theory of the range of bugs a student might have. Without such informal theories, the search space explodes combinatorially, leaving ACM with a hopeless task. The operators, shown in figure 1-3 are actually quite high level. For example, they assume that the digits are only subtracted within the *same* column, precluding the possibility of subtracting, say, the units digit of the subtrahend from the digit in the leftmost column of the minuend. In order to capture such behaviour, the operators would have to be encoded at a much lower level. ACM would have to include a subtraction operator which can be applied to *any* two numbers in the problem. Including such finer-grained operators would lead to an explosion of the already huge search space. The reader may feel that such a bug is so unlikely that it is not worth breaking-up the operators into finer-grained ones. Unfortunately, other domains are not so forgiving as multi-column subtraction. In a study of children's errors in fraction subtraction, Evertsz (1982) found that they do indeed apply operators to unexpected pairs of operands. For example, the whole number part of a mixed fraction is sometimes added to the numerator, and the numerators may even be multiplied by the denominators.

There is also another sense in which the initial information provided to ACM pre-empts the set of bugs to be modelled. The set of problems is user-generated, and implicitly encodes expectations about what problem features are likely to reveal bugs in a student's algorithm.

Summary

Viewed from the 'search' perspective, DEBUGGY and ACM are both searching for the path taken by the student; DEBUGGY's operators are just stated at a higher level. ACM's operators are very subtraction specific. In order to extend the coverage to bugs which have not been observed in advance (and may therefore have influenced the choice of initial operators), we will need to provide ACM with even lower level operators.

There is a trend in student modelling research to develop methods with greater coverage, but unfortunately this leads to greater search. Therefore, we need to find methods of reducing this search space, without sacrificing the greater coverage afforded by machine learning techniques.

1.7. Student Modelling as Inductive Inference

Student modelling is a process of 'induction', and as such is an uncertain process. Hume, some two hundred years ago, pointed out that there can be no 'valid' form of inductive inference. Induction is a process of 'jumping' from propositions to conclusions which imply more than is contained in the original propositions. For example, having seen many instances of the sun rising, we may feel justified in predicting that it will rise forever more. However, we can never *prove* that this is true; for all we know, the sun may not rise tomorrow. The problem is that induction is based on an assumption: 'What holds today, will hold tomorrow, all other things being equal'. This is a working assumption, without which no organism could adapt, having no basis for learning; but the assumption cannot be verified.

The maxim 'What holds today will hold tomorrow...', is implicit in the process of student modelling. On the face of it, it is valid to assume that the student will always respond in the same way to a given situation. The twist in this assumption is that the student is *never* confronted with the *same* situation. Human beings learn and forget, so no matter how careful the tutor is about holding the environment constant, the student's 'internal environment' will inevitably have changed in some small, and possibly significant way. This has important implications for how we model the student. We could iron out the twist in the above assumption, if it were possible to infallibly model the processes of change within the student - we can reject this notion for the forseeable future.

Even in a purely diagnostic context, where the tutor is not trying to teach anything new, the student's internal state will be affected by the diagnostic problems. An analogous notion exists in Quantum Mechanics, where the very act of observing an entity is said to affect its behaviour (Heisenberg's Uncertainty Principle). This makes differential diagnosis in ICAI very different

from that in other areas, such as electronic fault diagnosis. Diagnostic programs such as GDE (DeKleer and Williams, 1987) assume that the process of taking a measurement does not affect the device under scrutiny. When modelling the student, it is useful to make this assumption, but the diagnostic program should be capable of retracting it when it cannot find a consistent interpretation of the data.

1.7.1. The Fallacy of the Crucial Experiment

In preferring to assume that the student's performance is consistent, unless forced to do otherwise, we are adopting a particular meta-diagnostic stance: slips and bug migrations are less desirable ways of accounting for the student's behaviour. If we did not adopt this bias, then we could model any student by assuming that his/her model changes for each problem. Similarly, a CP (Critical Problem) does not entitle the diagnostic program to unequivocally reject the losing hypothesis. The losing hypothesis is still valid if it is augmented by, say, a slip hypothesis (i.e. that the student slipped up). Scientific research is faced with a similar dilemma; can a crucial experiment really discriminate between two competing theories? The following example (adapted from Copi, 1953) should convince the reader that, regardless of domain, there can never be a truly crucial experiment. An experiment is only crucial if the proponents of the two theories agree not to adopt post-hoc auxiliary theoretical postulates. Similarly, a CP is only a crucial experiment if the tutoring system cannot explain the results by adopting a slip hypothesis.

During the Renaissance, it was generally believed that the earth was flat. Two notable dissenters were Christopher Columbus and Nikolaus Copernicus, who subscribed to the spherical-earth theory. Columbus presented the following argument (Copernicus' was a slight variant, but essentially the same):

As a ship sails away from the shore, its deck is seen, by an observer on land, to disappear long before its masthead. Were the earth flat, the deck and masthead would disappear at the same time.

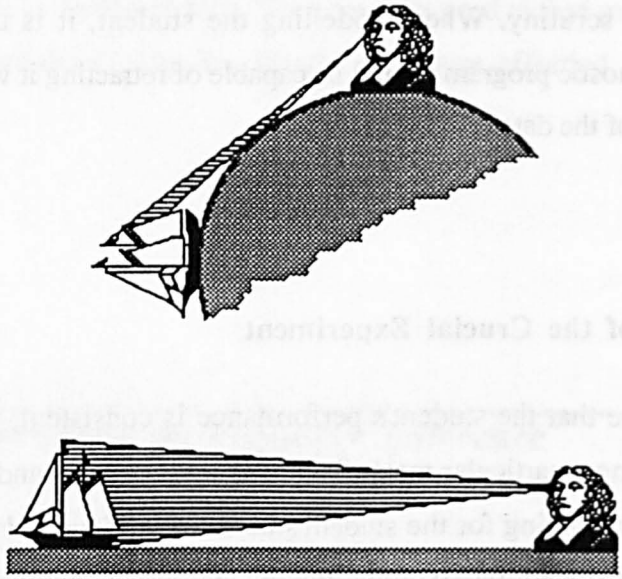


Figure 1-9 — Columbus/Copernicus vs. the Flat-earthists

So, the flat-earthists depart to perform the above crucial experiment, and are distressed to find that the deck vanishes before the masthead. A beautifully elegant, and certainly crucial, experiment you may feel. However, all is not lost for the flat-earthist; it is entirely possible to accept the results of this experiment, believe in a flat earth, and remain consistent. The Columbus/Copernicus thesis rests on an unstated, but important assumption: 'Light travels in straight lines'. But what if light follows a curved path? As you can see in figure 1-10, the experiment is no longer crucial.

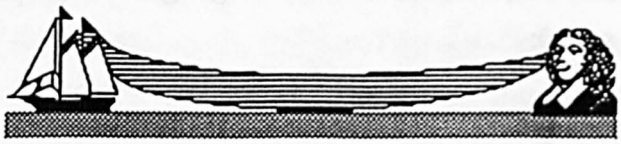


Figure 1-10 — The Flat-earthists Rationalise Damning Evidence

Columbus and Copernicus need *at least* one more crucial experiment, one which 'proves' that light travels in straight lines, before they can persuade the flat-earthists to relinquish their

position. Sadly, the next crucial experiment will suffer from the same limitation; all theories, and therefore experimental hypotheses, are based on assumptions.

1.7.2. Summary

Regardless of domain, there is no such thing as a truly crucial experiment. When diagnosing fairly consistent devices such as electronic circuits, it is expedient to assume that the measurements provide the desired information, and to ignore the option of adopting auxiliary hypotheses whose sole purpose is to enable us to hang on to our favoured hypothesis. For student modelling we cannot adopt this assumption without at least allowing for the possibility that the student's response is uninformative, because s/he made a slip. Thus, when we talk about a CP discriminating between two hypotheses, what we really mean is: the CP enables the tutoring system to *favour* one hypothesis because of the meta-modelling assumption that the student's behaviour is more likely to be consistent than inconsistent. Subsequent data may dictate that the rejected hypothesis be reinstated, because the student's behaviour (overall) is better characterised by the rejected model plus an assertion that the unfavourable response to the CP was just a slip. The CP-generation techniques developed in this thesis only allow one to increase the information obtained from diagnostic problems; they do not enable us to obtain *perfect* model-distinguishing data.

1.8. Previous Approaches to Problem Generation

The problem of generating hypothesis-testing examples has been tackled with a number of different goals in mind, from student modelling, to refining generalisations in machine learning (Mitchell, 1978). In this section, we review two methods developed with student modelling in mind.

1.8.1. IDEBUGGY: Using Pre-stored Problem Templates

IDEBUGGY (Burton, 1982) adopts an essentially heuristic method of generating CPs. Each primitive bug has an associated set of problem templates which, when instantiated, are likely to produce problems which reveal the presence of that bug. Not all of the problems so generated will have this feature, therefore the system needs to run the student model on the problem to make sure that it does the required job. The problem templates are instantiated until a discriminating problem is found. It may happen that the two competing models are so similar that the program fails to find an appropriate problem. In this case, IDEBUGGY looks to see if one of the problems, presented to the student earlier on, discriminates between the two hypotheses. If such a problem is found, the program tries to modify it by incrementing and decrementing some of its digits, and adding or deleting extra columns.

This heuristic approach is not guaranteed to find a CP if one exists. There comes a point where the system must decide whether to carry on looking, or assume that the two models are logically equivalent. It does this after 500 attempts, though it can always retract this assumption if a later problem reveals a difference between the two hypotheses.

Though quite effective, IDEBUGGY's approach has several drawbacks. The system implementor must specify the problem templates for each of the bugs in the database of models. This means that any changes to the models must be echoed in their associated problem templates. This reliance on an outside party to maintain the effectiveness of the problem generator means that one could not really extend the method to deal with student models generated using machine learning techniques.

Because the process is heuristic, it can only guess whether two models are equivalent. However, if two models are quite similar it is probable that they will behave identically most of the time, leading the system to erroneously deduce that they are equivalent. Ideally, conclusions about equivalence should be derived by inspection of the models themselves. Unfortunately, IDEBUGGY's representation of models does not lend itself to such an analysis. In section 1.4.4 we drew attention to the fact that the opacity of the representation meant that it could not predict in what way different parts of the skill can become bugged. This opacity also makes it difficult to reason about what the buggy procedures compute. One can imagine getting a handle on this problem by annotating the buggy procedures with an abstract description of the relationship between their inputs and outputs. However, this still requires that the problem-generation capabilities be manually maintained. It would be far better to

represent our models in a form which is open to automated inspection. The work described in the next section is a step in this direction.

1.8.2. A Rule-based Task Generation System

Sleeman (1981,1983) describes a problem generator for student models expressed as ordered production rules. This problem generator has been applied to the domain of algebra, but it is claimed (Sleeman, 1983) that it:

> "... is suitable for domains in which the objective is to transform an initial state into either a pre-specified (goal) state or at least one with specified characteristics."

In this section, we take issue with this claim, and attempt to show that it can only be applied to a very limited range of domains. In order to set this work in the context of production rule models in general, the next paragraph outlines the architecture of typical PSs.

Production System Architectures

The architecture of PSs varies a great deal (c.f. Newell (1973), Forgy & McDermott (1977), Anderson (1983)), nevertheless, there are four components common to them all: Working Memory, Production Memory, Recognise-act Cycle, and Conflict Resolution Principles. Working Memory is used to hold all of the intermediate computations of the system, and tends to have a short retention span. In contrast, Production Memory holds all of the system's permanent knowledge. This knowledge is represented as IF-THEN rules (productions), each consisting of an antecedent (termed the 'lefthand side' or 'LHS') and a consequent ('righthand side' or 'RHS'). The Recognise-act Cycle is part of the 'control structure' of the PS. On each cycle, the rules in Production Memory are compared with the contents of Working Memory. All rules, whose antecedents are 'satisfied' by Working Memory, are 'instantiated' and put in the Conflict Set. The Conflict Resolution Principles of the system represent the other half of the control structure. These principles are applied to the Conflict Set, so that the 'best'

instantiation is chosen. The chosen rule is said to 'fire', in other words, the changes to Working Memory, specified in the righthand side of the rule, are executed. This process is repeated ad infinitum, although in reality one's simulation model usually enters a 'halt' state, for example if no rules are satisfied by Working Memory, then no rules are eligible for firing, and the cycling stops.

Rule Classifications

For the purposes of problem generation, Sleeman classifies the LHSs of rules into the following two categories:

Potentially interacting: Two rules, **r1** and **r2**, can potentially interact if **r1** has the conditions **[C1...Cm,Cn...]**, and **r2** has the conditions **[...Cm,Cn...Cz]**. This is because, if the problem contained the pattern **[C1...Cm,Cn...Cz]**, then both rules would be eligible for firing. This definition of rule interaction seems puzzling, until one realises that Sleeman's use of the term 'condition' is at variance with that generally used in the production rule community. The term 'condition' is normally taken to mean an element in the LHS of a production rule. A typical LHS will have a number of condition elements, each capable of matching a range of possible working memory elements. Now, it is clear that the fireability of a rule is wholly dependent on the contents of Working Memory. If we imagine a situation in which Working Memory is initialised with all of the elements required to instantiate every rule in the ruleset, then all of the rules will be eligible for firing[1]. Therefore, all rules (apart from mutually exclusive ones) are potentially interacting - it all depends on what is in Working Memory.

Sleeman uses the term 'condition' to denote what is normally termed a 'condition sub-element'. This is because the rule LHSs of his models only ever contain one condition, and likewise Working Memory only ever contains one element. The problem generation algorithm applies to models containing single-element LHSs only; there is no provision for multi-element LHSs, or for negated condition elements (i.e. those which require that no matching element be present in Working Memory).

[1] With the following caveat: rules with mutually exclusive conditions can never be instantiated at the same time. This occurs where one rule specifies that some element be present in Working Memory, whilst the other rule requires that it be absent. This is the only case where rules are definitely non-interacting.

Non-interacting: These rules form the complement of the potentially interacting ones. For example, a rule with the condition **[C1,C2,C3]** cannot be instantiated at the same time as one with the condition **[C1,C2,C4]** (but as we have noted, only as long as there is only one working memory element).

Generating Problem Templates

The program uses templates to generate a problem which discriminates between two models. To do this it must first be provided with an initial string (i.e. problem answer), a *generative* version of each rule, and a set of terminating conditions (number of iterations, the order in which rules must be fired, and so on). The rule MULT, and its generative version are shown below (items in bold are variables; those in italics are segment variables, i.e. they can match a (possibly empty) sequence of items).

MULT
Comment: This rule matches a pattern with an embedded multiplication problem. For example, it would match (2 + 3 * 4). It alters the expression by replacing the embedded multiplication by its value (3*4 becomes 12).
Rule definition: if (*lhs* **M * N** *rhs*)
then (*lhs* [**M*N** evaluated] *rhs*)
Generative version: (*str1* **NUM** *str2*) ⇒ (*str1* **NUM * NUM** *str2*)

Note how similar the generative version is to the original rule definition; Sleeman does not say how the generative versions are derived. From this example, one would expect them to be machine-derivable, but the other rules do not seem to have the same simple relationship with their generative version. Sleeman does not actually provide further examples of generative rules. However, we can derive two more generative rules from the (MULT SOLVE FIN2) example (figure 1-11).

SOLVE
Comment: Solves for X by moving the numeric multiplicand over to the righthand side. For example, (5 * X = 25) becomes (X = 25/5). If M is zero, then the result is infinity.
Rule definition: if (M*X=N)
then (X=N/M) or infinity
Generative version: (NUM) ⇒ (1 * X = NUM)

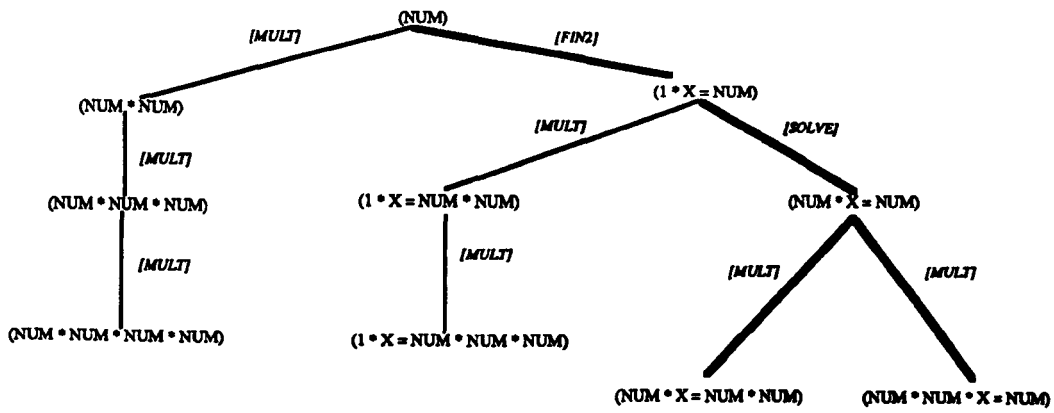


Figure 1-11 — Generation of a problem template for (MULT SOLVE FIN2)
Adapted from Sleeman (1981)

Figure 1-11 shows the system generating two problem templates for the model (MULT SOLVE FIN2). The user has supplied the initial state, and the termination condition that the rules be fired in the order: **MULT, SOLVE, FIN2.** The routes to the two templates have been labelled with heavy black lines.

FIN2
Comment: If the expression is of the form: (X = M/N) then either leave the answer as M/N[1] or replace M/N by its value.
Rule definition: if (X = M/N)
then (M/N) or [M/N evaluated]

To the extent that the system is capable of generating problem templates from a set of 'generative rules', it is an improvement over the solution employed in IDEBUGGY - that of

[1]The first action of the rule FIN2 is actually (M N). However, this appears to be a typographic error because Sleeman's examples of FIN2's execution imply that the action should be (M/N).

associating a set of pre-calculated problem templates with each bug. However, the algorithm has its problems. The system performs a breadth-first search on the rules in the model. For cases where there are more than a few rules in the models, the branching factor will lead to a combinatorial explosion of the search space. The example above relied on the user providing it with information about when to stop its search. The system on its own is incapable of deciding when to stop building the templates. The approach is a step in the right direction, but it does not overcome the drawback of having to incorporate hard-wired problem templates; its ability to derive a template is completely dependent on the user.

Instantiating Problem Templates

In order to set an actual problem, the system needs to instantiate its chosen problem template. Sleeman has not tackled this problem, and argues that (Sleeman, 1982):

> "This is not a very demanding task, but neither is it a very rewarding one..."

This is certainly true for the algebra models handled by his system. The problem templates only contain literals and unnamed variables, the latter always standing for numbers. It is indeed trivial to instantiate these unnamed variables with arbitrary numbers. However, this property of having completely unconstrained variables does not extend to other domains. Production rule models in domains such as multi-column subtraction (a domain which one tends to regard as 'simpler' than algebra), contain LHS conditions which constrain the relationships between variables in the problem statement. The following rule (taken from Young and O'Shea, 1981) illustrates this point. It specifies that a borrow must be performed when the current digit of the subtrahend is greater than that of the minuend. Thus, in setting a problem which requires borrowing, it is not sufficient to instantiate variables with random numbers; the numbers must satisfy the constraints specified within the production rule models.

Rule B2A: S > M => Borrow

This example highlights a limitation of the template-generation procedure, one which severely restricts its applicability to other domains: there is no provision for the encoding of constraints on variable values.

Discriminating Between Two Models

The problem of discriminating between two models is tackled by first classifying them into one of two cases:

(i) Both models contain the same rules, but a pair of 'potentially interacting' rules are differently ordered;

(ii) The models contain different rules.

In the former case, the system applies 'algorithm-A', whilst the 'different rules' case is handled by 'algorithm-B'.

Algorithm-A: Given two models, the system produces a problem template for each model, after removing one 'potentially-interacting' rule from the first model, and the other 'potentially-interacting' rule from the second. It then extends each problem template by using the deleted rule in generative mode. The extended template is accepted if the overlap pattern for the two interacting rules is contained in the template, and the template satisfies the two deleted rules. This algorithm must be applied to each conflicting pair in the two models.

Algorithm-B: Where the two models contain differing rules, the system classifies the two differing rules according to whether the conditions or actions are different. It can only handle models where there is only one pair of differing rules; furthermore, where *both* the condition and action parts differ, it treats the rules as if only the condition parts differ. Now, if the problem template for one rule does not satisfy the other, then that problem template discriminates. If this is not the case, then the algorithm fails. It cannot alter the problem templates so that they discriminate between the two rules. For cases where the action part of the two rules differs, the algorithm just assumes that problem templates will discriminate.

The Algorithm's Limitations

Sleeman's work is of importance because it represents the first attempt to generate problems on the basis of the *form* of the rules in the models, rather than on the basis of 'canned' procedures attached to each rule. However, it can only be applied to a very limited range of production rule models, and even then it only works when in partnership with the user (who must supply the termination condition).

It can only be applied to models where Working Memory only ever contains one element, and all of the rules have one condition element only. There is no provision for multiple LHS conditions, or negated condition elements. The algorithm also ignores the relationships between variables in the LHS. For example, it could not generate a discriminatory problem for the following two one-rule models.

Model-1
Rule definition: if (X X)
then (They are equal)

Model-2
Rule definition: if (X Y)
then (They are equal)

As far as it is concerned, the working memory element (3 3) is a perfectly valid discriminatory problem. Clearly, only problems where $X \neq Y$ will discriminate; but the program misses this because it ignores the implicit inter-variable constraint in **model-1**: the first variable must be equal to the second.

In the simple algebra examples, tackled by the program, inter-variable constraints can be ignored. Under such conditions, the program can generate a discriminatory template, where the two models contain the same rules, but ordered differently. In the more general case, where the two models contain different rules (limited to one pair only), the algorithm is incapable of generating a discriminating template.

In summary, Sleeman's approach is a step in the right direction, but has very limited applicability. It can only be applied to single-element Working Memories, where the two competing models contain the same rules (ordered differently), have a single LHS element, have no inter-variable constraints, and the user has supplied a termination condition.

1.9. Conclusions

We have argued that CP generation goes hand in hand with efficient diagnosis; this becomes even more true as tutoring systems search ever larger hypothesis spaces. The bug-library approach to student modelling has limited coverage, and this has led researchers to turn to machine learning techniques. However, increased coverage entails more search; therefore it seems likely that, in the future, there will be an even greater need for domain-independent CP-generation capabilities.

To date, there have been two attempts at tackling this problem. The first, IDEBUGGY, relied on the user to supply canned, bug-specific problem-generation procedures. Sleeman presents a problem-generation algorithm, and argues that it is domain-independent. However, our analysis reveals that its presuppositions are too restrictive for it to be generally applicable.

Chapter 2

Reverse Path Collection

2.1. Introduction

Our goal is to develop a domain-independent method for generating Critical Problems. The question of how the production rule models are derived in the first place is quite separate, and is not addressed in this thesis; we will assume that our program has access to a model-proposing system which uses its own expertise to offer up a pair of candidate models. We know from earlier work that this is a realistic assumption (e.g. Sleeman and Smith, 1981).

In this chapter, we argue that the key to CP generation lies in developing an abstract specification of the mapping between the inputs and outputs of the competing models. From the hypothesis-testing perspective, the intermediate processing performed by a model is only relevant in so far as it defines (procedurally) the mapping between inputs and outputs. The hypothesis-tester can only manipulate inputs and observe the student's outputs.

To represent our student models, we will develop a forward-chaining production rule language. We then develop a compiler (RPC) which, given a production rule model, attempts to produce an abstract specification of the mapping between its inputs and outputs. This it does by analysing the dependencies between the rules in the model. Though able to cope with some models, the compiler is not without its problems. In subsequent chapters, we will

develop an algorithm which overcomes these deficiencies and can handle more general forward-chaining production systems.

The algorithm outlined here shares with that of Sleeman (1982) the idea of generating a problem template by searching backwards from an output description. However, that is where the similarity ends; it overcomes all but one of the limitations of Sleeman's scheme, (it is unable to handle PSs containing loops). Unlike Sleeman's task generator, the Reverse Path Collector (RPC) can handle rules with multiple condition elements, named variables and match predicates, and does not rely on constraints such as the assumption that only one rule pair discriminates.

2.2. Diagnosis and the Methodology of Science

The problem of student modelling has a lot in common with that faced by science in general. Given that we accept that the scientific approach is of some practical value, it follows that a diagnostic program would benefit from adopting the methodology of science, where appropriate.

As with many questions, one's conception of the major goal of science is fairly personal. It may range from one of promoting the welfare of humankind, to solving particular applied problems such as developing more powerful nuclear weapons, which some would say is the antithesis of the former goal. Scientific pursuits can be divided into two schools: pure and applied. Pure science is concerned with developing our understanding of the world, while the ends of applied science are purely practical. However, a common thread runs through all of these views of science: the primary aim of science is the advancement of knowledge. The goal of the diagnostic component of a tutoring system has more in common with that of applied than pure science. Unlike the more general pursuit of cognitive modelling, where the knowledge acquired is reward enough, student modelling is concerned with obtaining *just enough* information to guide its teaching - anymore is a hindrance (if the system tries to model at a finer level of detail than is necessary, then it will be expending wasted energy trying to characterise irrelevant minutiae).

The accretion of scientific knowledge is not merely a process of data accumulation, science seeks to provide more than just a description of phenomena which have been observed in the past; it produces a conceptual reconstruction of the observed phenomena which goes further than such data, and maps out the conditions which determine how similar objects will behave in the future. Such scientific generalisations take many forms, from collections of mathematically-stated laws to complex causal models, but we shall informally group them all under the umbrella-term 'theory'. Theories are evaluated by formulating experimental hypotheses, with which the scientist makes specific predictions about the behaviour of objects in the theory's domain of application. To be of any use, the experimental hypothesis has to be stated in a way that can be tested. For example, whereas a causal model such as the kinetic theory of gases provides us with a conceptually satisfying reason why increasing the temperature of a gas, with volume held constant, increases the pressure (theory-1), the level of description is not well suited to the task of devising a crucial experiment which will decide between this model and one based on mutually repelling molecules (theory-2).

Theory-1: What we measure as an increase in temperature of a body of gas is actually an increase in the mean kinetic energy per molecule; the mean velocity of a molecule of gas is determined from the mean kinetic energy per molecule; thus, the increase in kinetic energy produces an increase in the velocity of motion of the molecules. Since the molecules of gas are prevented from travelling further by the vessel of constant volume, they strike the inside surface of the vessel more frequently, and so increase the pressure on the walls.

Theory-2: Gases are made up of mutually-repelling molecules, where the force is inversely proportional to the inter-molecular distance. Increasing the temperature produces a proportional increase in the repelling force (proportional to the cube of the molecule's radius). Gas molecules exert a repelling force on other objects as well (e.g. the enclosing vessel).

We can ease the problem of comparison by re-encoding the theories into a law-like formulation. Theory-1 becomes the General Gas Law $PV = nRT$, while theory-2 can be summarised by the following formula: $PV = nm^3T$, where **P** is its pressure, **V** the volume, **n** the number of moles, **R** its gas constant, **m** the average molecular radius, and **T** its temperature. By converting a deeper causal model into a simple universally quantified law, we

derive a description which contains either measurable variables or constants (compare this with the variables in the causal model, e.g. the velocity of motion of the molecules). The variables and constants summarise the unobservable processes of the causal model (sometimes termed 'intervening variables').

The above principles apply equally well to the problem of testing the accuracy of a student model. The production rule model can be viewed as playing the role which the molecular model served in the above example. Although it is executable, and can generate predictions about specific situations, when two such models are compared, it is not clear under what conditions they would produce different behaviour. In other words, the representation, as it stands, is of little use in formulating an informative experiment. Thus, our goal is to find a representation which is both equivalent (in terms of its input/output behaviour) to the original, well suited to inter-model comparison, and instantiable, i.e. models expressed in this new representation language can make predictions about behaviour in specific situations. Like the General Gas Law, the new representation will only contain terms which represent the variables which can be manipulated or measured (conventionally termed the 'independent' and 'dependent' variables, respectively). In the context of student modelling, the independent variable is the problem statement, and the dependent variable any machine-observable behaviour produced by the student. The new language should compile out the intervening processes of a production system (such as pattern matching, and the recognise-act cycle), yielding a description of the mapping between inputs and outputs only.

We will now describe the production system architecture to be used in this study.

2.3. The Production Rule Language

In designing a production rule language for representing student models, we are faced with a large space of architectural options. Conflict resolution strategies include rule order (c.f. PSG: Newell, 1973), recency and specificity (c.f. OPS2: Forgy and McDermott, 1977), and goal-oriented restrictions (c.f POPSI, Evertsz (1984)). Rules can fire serially, or even concurrently

(c.f. Thibadeau, Just and Carpenter, 1982). Working Memory can be represented as a set of elements, or as activations in a declarative network (c.f. ACT*, Anderson, 1983), and it may or may not be possible to delete items from it. The logical infrastructure of rules can also vary greatly from mere conjunctions of terms consisting of constants and variables, to arbitrarily nested combinations of **NOT**, **AND**, **OR**, and match predicates (c.f. COPS: Motta et al (1986)).

The jury is still out on the question of what constitutes the 'best' architecture for student modelling, and, in the author's opinion, will remain so indefinitely; it is very likely that domains with unusual features will be best suited to a production rule language especially created for the job. Thus, the choice of language for generating CPs is somewhat arbitrary. A simple ordered PS is a universal computer (Anderson, 1976), and so can compute (i.e. be used to model) anything which any other interpreter can. However, ordered rules have very little status as independent entities; what does it mean to say that the student has all of the correct rules for a task, but two of them are incorrectly ordered? With unordered PSs we can remove rules, and plug new ones in, without worrying about where they should be inserted. In the author's opinion, this makes them preferable for student modelling. By adopting this position, we do not lose anything; the problem generation techniques, developed in this thesis, apply equally well to ordered PSs (ordering rules is a relatively crude method of incorporating the notion of rule specificity, as used in OPS-like interpreters).

In this study, we shall define a language ('PGPS'; Problem Generator Production System) which is a slightly modified version of the OPS family of interpreters (Forgy and McDermott, 1977). The pattern matching facilities are similar to those of OPS interpreters, but have been extended to allow user-defined match predicates. These predicates are defined in Lisp, and evaluate to either 'true' or 'false' (more specifically, 'T' or 'NIL'). A description of PGPS's syntax can be found in Appendix I.

BNF-like definitions, such as those found in Appendix I, are designed to disambiguate the syntax of rules. However, they are not well suited to imparting a feel for what typical rules look like. The following section paraphrases the formal definition of syntax, and presents a simple PGPS rule. This is followed by a definition of PGPS's conflict resolution strategy, and a scenario in which a simple ruleset sums the even numbers from **2** to **5**.

2.3.1. An example PGPS rule

A PGPS rule consists of a rule name, a lefthand side (LHS) and a righthand side (RHS). The LHS is delimited by the symbols **IF** and **THEN**. The symbol **THEN** denotes the beginning of the RHS. An LHS is made up of one or more LHS terms, conjoined by ampersands (&). Each LHS term can be either a Working Memory pattern (possibly negated), or a function call (termed a 'match predicate') returning **T** or **NIL**. Function calls are identified by the fact that their first element is prefixed with an asterisk. A precondition for instantiating a rule is that all of its match predicates return **T** for the current set of arguments.

The following LHS is a conjunction of three terms. The first contains the variables **?x** and **?y**, and matches any Working Memory element (WME) of that form. The second specifies that there should be no element of the form: **(already subtracted)** in Working Memory. The third term is a match predicate which requires that **?x** be greater than **?y**.

LHS: (subtract ?y from ?x) & ~(already subtracted) & (*greaterp ?x ?y)

RHSs are similar to LHSs, except that they cannot contain a negated pattern, and function calls can return a value other than **T** or **NIL**. There are also two special functions: ***OUTPUT** and ***HALT**. The function ***OUTPUT** signifies that its argument should be sent to the outside world (i.e. it is behaviour which can be observed by an external entity), while ***HALT** stops the execution of the interpreter. The following RHS outputs the result of subtracting **?y** from **?x**, and deposits the element **(already subtracted)** to stop the rule from firing again.

RHS: (*output (*subtract ?x ?y)) & (already subtracted)

Combining the LHS and RHS with the name, **SUBTRACT**, gives the following rule:

```
(define-rule subtract
 if (subtract ?y from ?x) & ~(already subtracted)  & (*greaterp ?x ?y)
 then (*output (*subtract ?x ?y)) & (already subtracted))
```

2.3.2. PGPS's Conflict Resolution Strategy

PGPS applies the conflict resolution principles 'refractoriness', 'recency', and 'specificity' (in that order), to filter the conflict set down to a singleton. If, after applying these three principles, more than one instantiation remains, then the ruleset is non-deterministic, and the interpreter generates an error.

Refractoriness

On each recognise-act cycle, a conflict-set of relevant instantiations is generated. The 'refractoriness' principle, eliminates all instantiations which have fired on previous cycles. This ensures that a production rule does not fire more than once to the *same* subset of Working Memory. By 'same' one does not mean 'equal' - the production rule, below, could fire continuously, even though it had already fired when matched to an *equal* element, on a previous cycle. This rule adds the element **(loop forever)** which triggers the rule again on the next cycle.

```
(define-rule loop-forever
 if (loop forever)
 then (loop forever))
```

This rule merely reasserts the element with which it matches. For those familiar with Lisp, the distinction here between 'same' and 'equal' is analogous to the distinction in Lisp between the functions **EQ** and **EQUAL**. Two lists are only **EQ** if they occupy the same location in memory (i.e. are one and the same entity); but two lists are **EQUAL** (as defined recursively) if their **CAR** and **CDR** are either **EQ** or **EQUAL**.

Recency

If, having applied the refractoriness principle to the conflict set, more than one instantiation remains, then 'recency' is used to reduce the set further. Given two instantiations, one will be preferred if it matches more recent elements in Working Memory (i.e. those added to Working Memory more recently). The following algorithm is applied to score the recency of an instantiation:

(i) compare the most recent WMEs, matched by the competing pair of instantiations;

(ii) if one is more recent than the other, then choose that instantiation;

(iii) if they are both equally recent, then discard that pair of WMEs, and select the next most recent WMEs matched. Go to step (ii);

(iv) if either (but not both) lists of WMEs is null, then return the non-empty instantiation;

(v) if both lists of matched WMEs are exhausted, then stop - both instantiations are equally recent.

At this juncture, it is worth explaining how a WME gets a recency value. For any series of cycles, the elements deposited in a later cycle are more recent than those from an earlier cycle. Where two or more elements have been deposited during the same recognise-act cycle, by some RHS, the elements later in the RHS are said to be more recent than those earlier. This is because RHS actions are carried out serially, earlier actions being executed before later ones.

Specificity

This principle is used to choose the more specific instantiations in the conflict set. Perhaps ideally, some rule, **r1**, can be said to be more specific than some rule, **r2**, if the set of **r1**'s possible satisfying Working Memories is a subset of those of **r2**. In general, it is not feasible to compute this relation. For example, which of these is the more specific, **r1** or **r2**?

LHS of r1: (?a) & (*evenp ?a))

LHS of r2: (?a) & (*primep ?a))

From our knowledge of the domain of natural numbers, we would say that **r2** is more specific, because there are more even numbers than prime numbers. For some predicates, this question may be undecidable. With this in mind, we choose the following definition. A rule, **r1,** is more specific than another, **r2,** if they match the same WM subset and either:

(i) the LHS of **r2** matches that of **r1** but not vice versa,

(ii) **r1** has more constraints (including equality ones) than **r2,**

(iii) **r1** has more negated patterns,

(iv) some pairwise matching of the negated patterns succeeds for **r2** matched with **r1** but not vice versa.

2.3.3. A PGPS ruleset for summing even numbers ‖

The following ruleset counts from **?x** to **?y,** and then outputs the sum of those numbers which are even.

```
(define-rule start-counting
 (* If you are summing from ?x to ?y, and you haven't counted-out ?x, then do so.)[1]
 if (sum from ?x to ?y) & ~(number ?x)
 then (number ?x))

(define-rule count
 (* If you are summing up to ?y, and you haven't got there,
    then deposit the successor of the current number.)
 if (sum from ?x to ?y) & (number ?previous) & ~(number ?y)
 then (number (*add1 ?previous)))

(define-rule start-summing
 (* If you haven't started working out the sum, then initialise it to zero.)
 if ~(partial-sum ?)
 then (partial-sum 0))

(define-rule sum
 (* If there is an even number in Working Memory, then add it to the partial sum, output the
   new partial sum, and note that you've used the even number just added.)
 if (partial-sum ?sum-so-far) & (number ?n) & ~(used ?n) & (*evenp ?n)
 then (partial-sum (*plus ?n ?sum-so-far))
   & (*output (*plus ?n ?sum-so-far))
   & (used ?n))

(define-rule halt
 (* Having used the number you started counting from, there is nothing else to do, so halt.)
 if (sum from ?x to ?y) & (used ?x)
 then (*halt))
```

If we were to run this ruleset with an initial Working Memory of **((sum from 2 to 5))**, then the rule **start-counting** would fire first, followed by three firings of **count**. At this point, Working Memory would contain five elements (in descending order of recency):

```
((number 5) (number 4) (number 3) (number 2) (sum from 2 to 5))
```

The rule **start-summing** would then take over, and deposit **(partial-sum 0)**. This would trigger **sum**, first on **(number 4)**, and then on **(number 2)**; the ordering being due to the fact that **(number 4)** is more recent than **(number 2)**. Finally, **halt** would bring the computation to a close, with the following outputs and final Working Memory:

[1] All comments are encompassed in parentheses and preceded by an asterisk, i.e. are of the form: (* ...). Note that there is a space between the * and the ensuing atom; if there is no space the asterisk/atom combination is a function expression. For example, the expression (* atom) is a comment, but (*atom) is a function expression.

Outputs: 4, 6.

Working Memory: ((used 2) (partial-sum 6) (used 4) (partial-sum 4) (partial-sum 0)
(number 5) (number 4) (number 3) (number 2) (sum from 2 to 5))

So, the sum of the even numbers between **2** and **5** is **6**. Note that the partial sums were calculated by invoking the Lisp functions ***plus** with the instantiated arguments **?n** and **?sum-so-far**. Similarly, the numbers between **2** and **5** were generated by the function ***add1**, which computes the successor of its argument.

2.4. Behavioural Equivalence

At the beginning of this chapter, we looked at the kinetic theory of gases and illustrated how much easier it is to design an experiment if we have a simple description of the relationship between the independent and dependent variables. Once we have such a description, the specific internal workings of the models can be ignored. For the purposes of hypothesis testing, we are only interested in how the model's behaviour is affected by the manipulation of the independent variables. If the two models produce the same behaviour, no matter how we vary the independent variables, then they are 'behaviourally equivalent', and no discriminatory experiment exists. Of course, we may be able to discriminate between them if we can develop an experimental technique which renders other, previously invisible behaviours, visible (for example, if we could *directly* observe the gas molecules). However, to achieve this we would have to redefine what is observable. We will assume that the student models rigidly define what behaviour can be observed; the system needn't consider the possibility of gaining access to intermediate variables which are not explicitly tagged as accessible. Thus, if two models are equivalent in terms of their outputs for a given class of inputs, then the system can conclude that the class of inputs cannot be used to generate an informative experiment.

‖2.4.1. Milner's Notion of Equivalence

Milner (1980) has developed a formalism for describing concurrent systems: CCS (Calculus of Communicating Systems). A key concept in his work is that of 'observation equivalence'. In this definition of weaker equivalence, the internal workings of the system are ignored; two systems are equivalent "... iff in all contexts they are indistinguishable by observation" (pg 7, Milner, 1980). Milner also defines other, stronger, levels of equivalence such as 'direct equivalence', where the internal actions of the systems must be identical.

Our notion of behavioural equivalence has much in common with Milner's observation equivalence. However, we are not particularly concerned whether two models are indistinguishable in *all* contexts. Our focus is on the problem of generating an experiment (CP) which discriminates between the two systems. Thus, we are concerned with the *classes* of input for which two systems are not behaviourally equivalent. CP generation involves an *existence* proof, rather than the proof of a universally quantified theorem. The program need only be satisfied that there exists an input for which the two programs are *not* behaviourally equivalent.

‖2.4.2. The Key Lies in Input/Output Mappings

How can we obtain a description of the mapping between inputs and outputs of a production rule model, which provides us with the same economy of description which the General Gas Law provided over the kinetic model? To gain a feel for what is required, consider the following two models for subtracting two numbers. The first, **SUBTRACT**, correctly handles problems where the minuend is less than the subtrahend, by prefixing the answer with a minus sign. The second model, **ABS-SUBTRACT**, contains a buggy rule which just swaps the two numbers, without prefixing the problem with the inverse sign (i.e. it computes the absolute difference of the two numbers).

Model: **SUBTRACT**

```
(define-rule negative-result
 (* If the minuend is less than the subtrahend, then swap them and
      prefix the new problem with a minus sign.)
 if (?minuend - ?subtrahend) & (*lessp ?minuend ?subtrahend)
 then (- ?subtrahend - ?minuend))

(define-rule positive-result
 (* If the minuend isn't less than the subtrahend, then the result will be positive.)
 if (?minuend - ?subtrahend) & ~(*lessp ?minuend ?subtrahend)
 then (+ ?minuend - ?subtrahend))

(define-rule subtract
 (* If you know what sign the answer will have, then just do the subtraction.)
 if (?sign ?minuend - ?subtrahend)
 then (?sign (*subtract ?minuend ?subtrahend)))

(define-rule halt1
 (* Having worked out the signed answer, output the sign and the number, and stop.)
 if (?sign ?number)
 then (*output ?sign) & (*output ?number) & (*halt))
```

Model: **ABS-SUBTRACT**

```
(define-rule positive-result
 (* Exactly the same as the rule in the above ruleset (SUBTRACT).)
 if (?minuend - ?subtrahend) & ~(*lessp ?minuend ?subtrahend)
 then (+ ?minuend - ?subtrahend))

(define-rule swap-numbers
 (* If the minuend is less than the subtrahend, then just swap them (this is a buggy rule).)
 if (?minuend - ?subtrahend) & (*lessp ?minuend ?subtrahend)
 then (+ ?subtrahend - ?minuend))

(define-rule halt2
 (* If the minuend isn't less than the subtrahend, then output the difference between them, and
halt.)
 if (?sign ?minuend - ?subtrahend) & ~(*lessp ?minuend ?subtrahend)
 then (*output ?sign) & (*output (*subtract ?minuend ?subtrahend)) & (*halt))
```

Both models divide the space of inputs (problems) into two disjoint sets, one where the minuend is less than the subtrahend: **(*lessp ?minuend ?subtrahend)**, and the other where it isn't: **~(*lessp ?minuend ?subtrahend)**. If we combine these constraints with the abstract WME which forms the input to each model, we can derive an abstract (i.e. one containing variables, rather than concrete numbers) description of each class of inputs. We will call this abstract description an 'Input Specification'. The two input specifications produce the following abstract outputs:

Model: **SUBTRACT**
Input Specification 1: (?minuend - ?subtrahend) (*lessp ?minuend ?subtrahend)
Output: - (*subtract ?subtrahend ?minuend)

Input Specification 2: (?minuend - ?subtrahend) ~(*lessp ?minuend ?subtrahend)
Output: + (*subtract ?minuend ?subtrahend)

Model: **ABS-SUBTRACT**
Input Specification 1: (?minuend - ?subtrahend) (*lessp ?minuend ?subtrahend)
Output: + (*subtract ?subtrahend minuend)

Input Specification 2: (?minuend - ?subtrahend) ~(*lessp ?minuend ?subtrahend)
Output: + (*subtract ?minuend ?subtrahend)

From the above input/output descriptions, one can infer that the CP must be of the form: **(?minuend - ?subtrahend)**, where **?minuend < ?subtrahend.** The critical difference between the two models is the sign of the answer. In general, the difference may be more complex than this. For example, if **ABS-SUBTRACT** *added* the minuend to the subtrahend, instead of subtracting them, then we could discriminate between the two models on the basis of the numeric part of the answer, regardless of sign. In the latter case, the important processes within the production system are the alterations performed on the input numbers. To handle such cases, one must analyse the flow of data from the input element(s), through the variables in the rules, to the output element(s).

Let us recap; the simple **SUBTRACT** vs. **ABS-SUBTRACT** example revealed two kinds of model-distinguishing output. The first is where the model outputs a pattern which is not *directly derived* from variables within the input elements (the +/- sign is an example of such an output; it is an RHS constant rather than a variable derived from variables in the problem statement). The second type is one which is derived by a series of transformations performed on the input variables (e.g. the expression **(*subtract ?minuend ?subtrahend)** is an abstract description of the transformation ***subtract** applied to the variables **?minuend** and **?subtrahend**). Therefore, our model-analyser should look for both types of output. Furthermore, the set of outputs is governed by the set of input specifications; in particular, constraints on the variables in the input elements determine what is output (the predicate ***lessp**, in the **SUBTRACT/ABS-SUBTRACT** models, is an example of such a constraint).

In general, a running model can follow a number of different paths to a halt state, depending on what the initial input looks like. We can say that the input specification defines the particular *path* taken through the space of possible routes from start state to halt state. If we can characterise the set of potential paths for a particular model, and collect the outputs along each of these paths, then we will have obtained a description of the mapping between inputs and outputs. In this thesis, we will move through a series of algorithms for exploring abstract paths in a production rule model. The remainder of this chapter presents a path-exploration algorithm which processes the rules in the reverse direction to that in which they normally run. This algorithm is only suitable for production rule models with very restricted properties, and represents the author's first attempt at an implementation of a CP generator. It is presented here in order to show why the approach is not sufficiently general to cover typical PSs, and thereby to motivate the more complex algorithm developed in Chapter 3.

2.5. The Reverse Path Collector (RPC)

2.5.1. The Role of Pattern Matching in a Production System

In production systems, the patterns in the LHS serve to define the conditions under which the rule is eligible for firing. These conditions are either satisfied by the initial contents of Working Memory, elements deposited by the RHSs of rules, or a combination of both. Thus, RHS actions and initial WMEs serve to trigger rules in the PS, and the LHS patterns of a rule define the set of actions which can trigger that rule. In other words, the instantiation of LHS patterns is *dependent* on the RHS actions and WMEs.

The particular form (pattern) of an RHS action determines which LHS patterns it is capable of satisfying. For example, the RHS action **(partial-sum ?n)** can be matched by the pattern **(partial-sum ?a)** or **(partial-sum 0)**, but can never be matched by **(number ?x)**. By analysing the RHSs of the rules in a ruleset, we can determine which actions a given LHS pattern depends on. Once we have determined all of the LHS/RHS dependencies of a ruleset, we can discard the actual LHS and RHS patterns and just keep a record of the dependencies amongst the variables in the patterns. This is because the process of pattern matching in a production system serves two purposes:

(i) to determine whether the conditions are right for a rule to fire (i.e. pattern matching defines the 'fireability' of rules), and

(ii) to bind the variables in the LHS, so that their values can be passed to the RHS actions, and then on to the LHSs of the rules which depend on those RHS actions, and so on.

Our goal is to obtain a description of the outputs of a ruleset - we don't care about purpose- (i) of pattern matching as it only served to define the dependencies between LHS and RHS patterns. However, we *are* interested in its second function (ii), because the outputs may contain variables. We need to know where these variables get their values from. The following simple ruleset illustrates these ideas.

```
(define-rule likes1
 if (?x likes-to-eat candy)
 then (?x likes sweet things))

(define-rule likes2
 if (?x likes-to-eat lemons)
 then (?x likes sour things))

(define-rule tolerates
 if (?x likes sour things) & (?x likes sweet things)
 then (?x tolerates (sweet and sour chicken)))

(define-rule halt
 if (?x tolerates ?food)
 then (*output (?x will eat ?food)) & (*halt))
```

In this example, the LHS of the rule **halt** depends on the RHS of the rule **tolerates**, which in turn depends on the RHS of the rules **likes1** and **likes2**. Once we have noticed these dependencies, we need only worry about those parts of the LHS/RHS patterns which affect the variables within the final output element: **(*output (?x will eat ?food))**. Other features, such as the fact that the tail of the first LHS pattern of the rule **tolerates** contains the constants **(likes sour things)**, are of no interest; the tail of the pattern only served to define what RHS actions the whole pattern could match with. However, the variable **?x** is important, because it appears in the final output of the model. Similarly, the binding of **?food**, within **halt**, is of interest to us.

In sum, our approach will be to analyse the dependencies amongst the rules in the model, compile-out the redundant aspects of the pattern matching process, and derive a description of the output patterns only.

2.5.2. PS Dependency Networks

The dependencies amongst LHS and RHS patterns can be defined as a directed (possibly cyclic) graph, with RHS actions linked to any dependent LHS patterns. We will call such a graph a 'PS Dependency Network'. A PS Dependency Network consists of a collection of linked nodes of different types. A network node can be either an LHS pattern, an RHS action, or a rule name. Each rule can be viewed as a sub-network of nodes. The RHS actions form the roots of the sub-network and have the rulename node as their only child, which in turn has the LHS patterns as its children. The following sub-network represents the rule **halt**, presented in the previous example[1]:

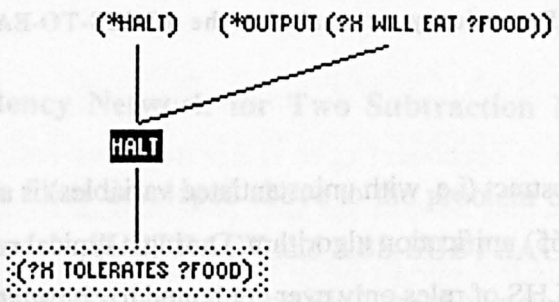


Figure 2-1 — Sub-network for the rule HALT

By linking the RHS nodes of a sub-network with any *unifying* LHS nodes, we can build the whole dependency net as shown in figure 2-2. Note that an RHS node can be linked to an LHS node of the same sub-network (i.e. of the same rule). However, none of our examples will include such dependency loops, as reverse path collection is flawed even without trying to handle such features.

[1]To improve readability, LHS patterns are surrounded by a dotted-box; rule names are printed in white on black; RHS patterns have no distinctive markings.

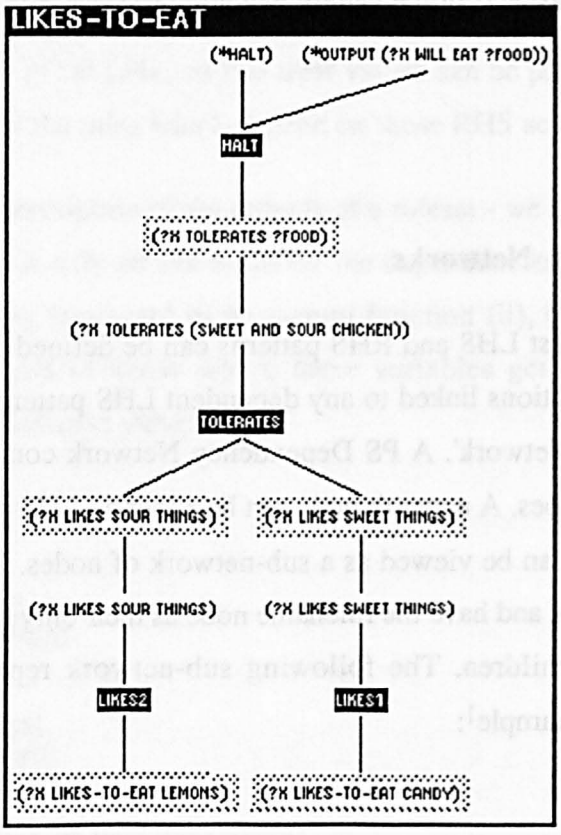


Figure 2-2 — Dependency network for the LIKES-TO-EAT ruleset

Pattern matching in the abstract (i.e. with uninstantiated variables) is achieved by employing a variant of Robinson's (1965) unification algorithm. During a typical run of a production rule model, the variables in the LHS of rules only ever get bound to *constants* in working memory. For example, the rule **halt** would match elements such as **(john tolerates tofu)** and **(mary tolerates (sweet and sour chicken))**. However, it is never called upon to match an element like **(?x tolerates ?y)**; typical forward-chaining production systems employ a one-way pattern matching process - variables are matched with constants only.

To compute the Dependency Network in figure 2-2, we had to match variables with variables (e.g. the **?x** of **halt**'s LHS with the **?x** of **tolerates**' RHS). Cases will also arise where a constant has to be matched with a variable, or a function expression has to match with a variable or constant (and vice versa). This is the reverse of the normal mode of matching, where variables match against constants. A production system never has to match a constant in the LHS with a variable in working memory. This two-way pattern matching process, termed

'unification', is due to Robinson (1965). For our purposes, Robinson's unification algorithm has been generalised to allow the unification of function expressions such as **(*subtract ?minuend ?subtrahend)**. Without this extension, the program would be unable to make sense of the expressions computed in the RHS of rules (see example below).

```
(define-rule r1
 if (goal is to add ?x and ?y)
 then (answer is (*plus ?x ?y)))

(define-rule r2
 if (answer is 5)
 then (*output (the answer is five)) & (*halt))
```

The RHS of **r1** is capable of triggering the rule **r2**; depending on the runtime value of **?x** and **?y**, **(*plus ?x ?y)** may match **5**. Therefore, in computing the possible dependencies between **r1** and **r2**, the program should treat the expression **(*plus ?x ?y)** as if it were a *variable*, and bind it to **5**.

2.5.3. A PS Dependency Network for Two Subtraction Models

We will now apply the ideas developed above to the problem of generating an abstract description of the two models, **SUBTRACT** and **ABS-SUBTRACT**, introduced in section 2.4.2. The Dependency Networks, generated by RPC, are shown in figures 2-3 and 2-4.

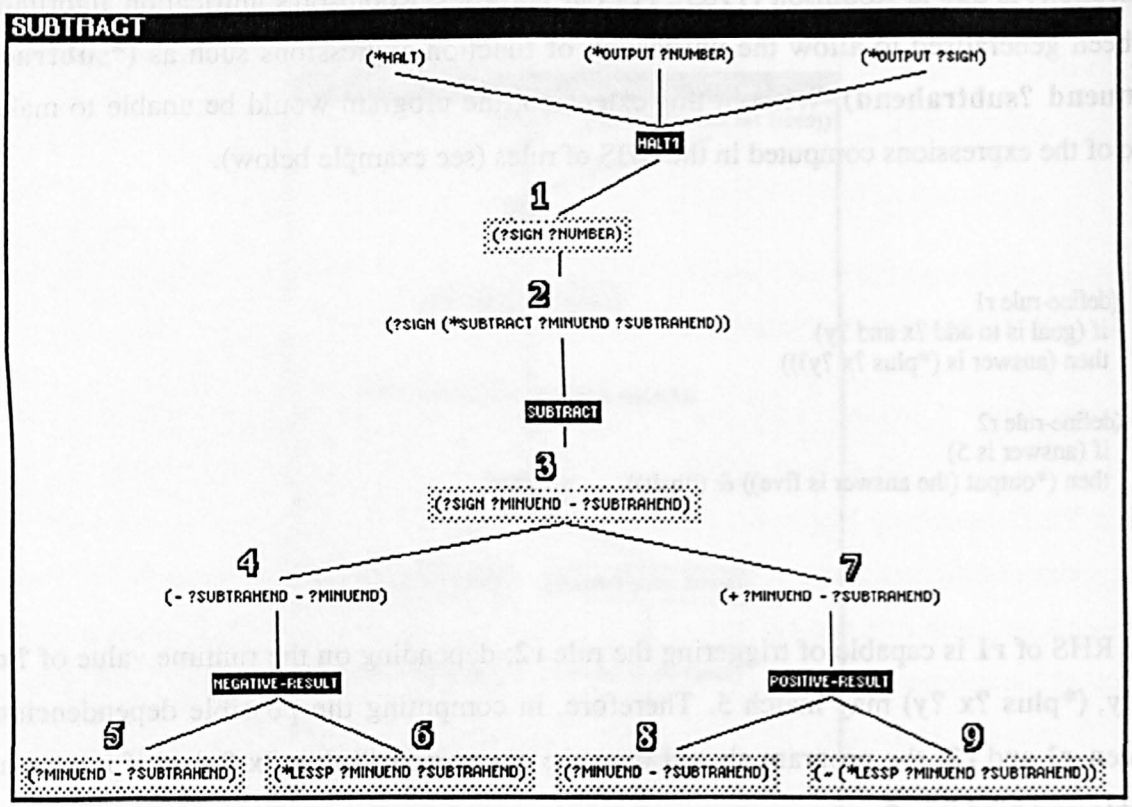


Figure 2-3 — SUBTRACT dependency network

The **SUBTRACT** Dependency Network shows graphically that the rule **halt1** is dependent on **subtract** which in turn is dependent upon *either* **negative-result** or **positive-result**. This is a disjunctive dependency because the LHS pattern of subtract unifies with the RHS patterns of both **negative-result** and **positive-result**, and could be triggered by either at runtime.

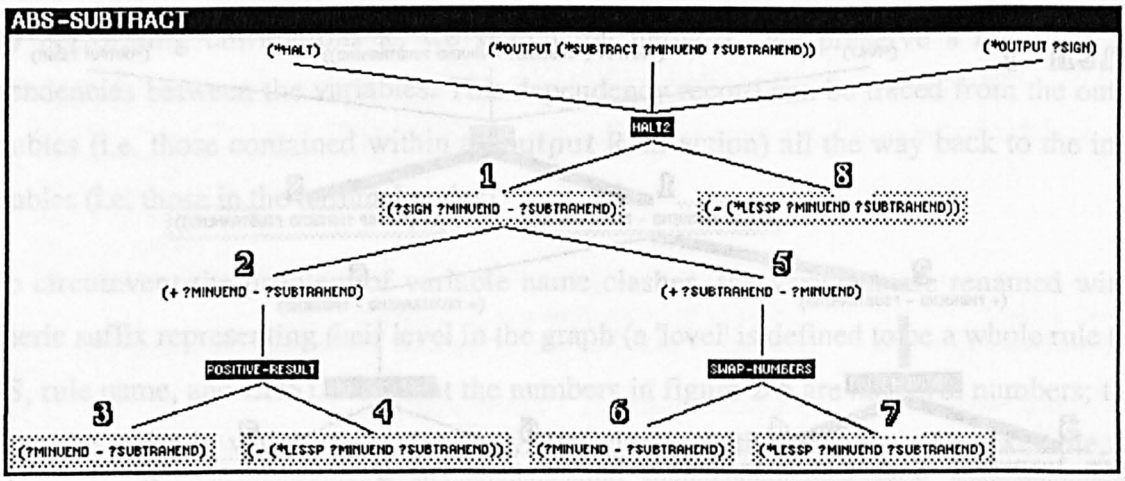


Figure 2-4 — ABS-SUBTRACT dependency network

The Dependency Network for **ABS-SUBTRACT** is a little simpler, there being only two levels to the graph. The rule **halt2** can be triggered by either **positive-result** or **swap-numbers**. By searching backwards from the halt rule (i.e. the one containing **(*halt)**), we can cover the whole graph and collect the set of possible paths through the production system. The actual search strategy is unimportant, as long as it systematically searches the whole graph. If we adopt a depth-first strategy on the **ABS-SUBTRACT** network, then it would be searched as shown in figures 2-3 and 2-4. The nodes in the figures are numbered in the order in which they would be visited (leaving out the rulename nodes, as they only form a redundant link between a rule's LHS and RHS).

As we noted earlier, the outward links from node 1 in the network form a disjunction, so there are two possible paths through the network (see figure 2-5). The first consists of the nodes: **(1,2,3,4,8)** and the second: **(1,5,6,7,8)**. The **SUBTRACT** network succumbs to similar analysis (figure 2-3), and yields the two paths: **(1,2,3,4,5,6)** and **(1,2,3,7,8,9)**.

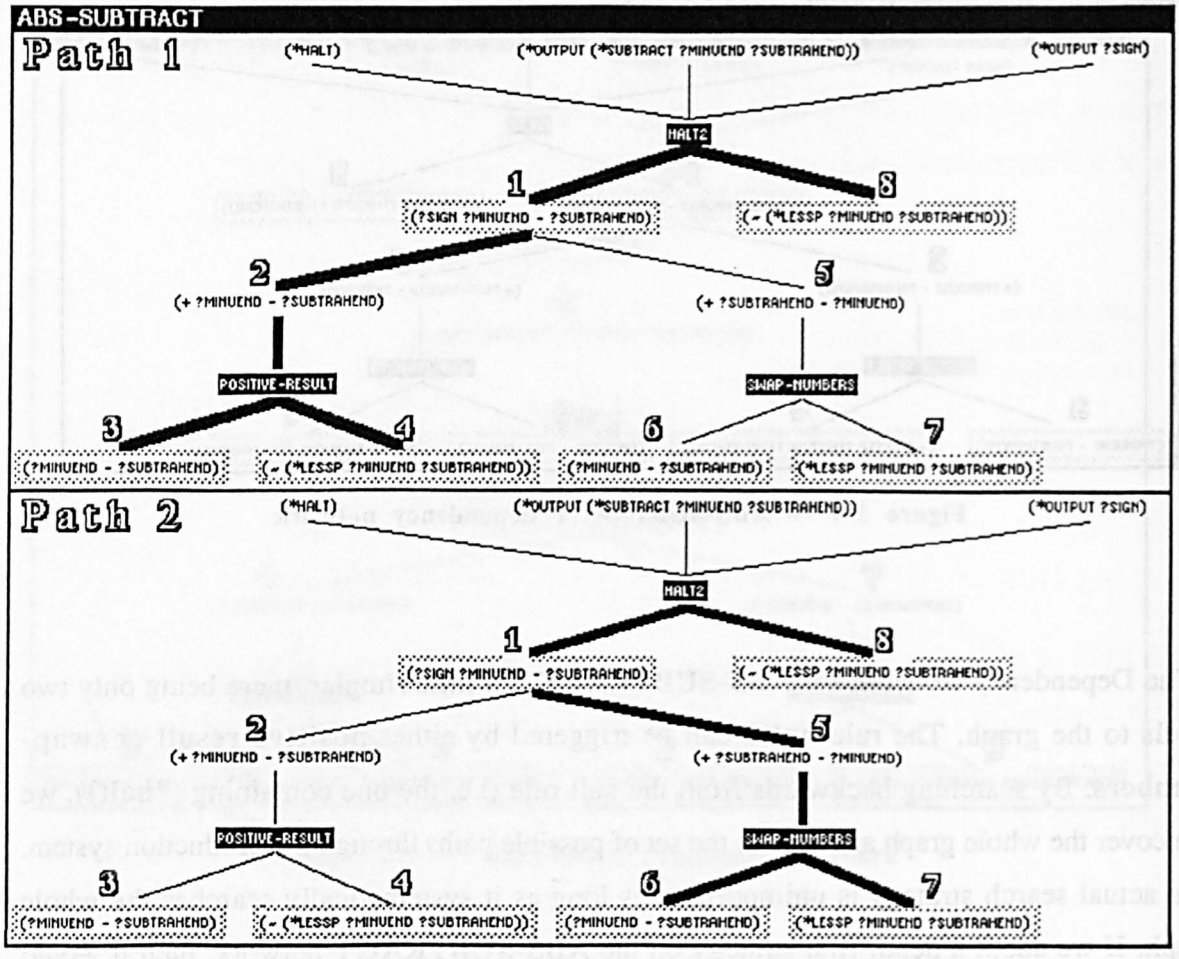


Figure 2-5 — Paths 1 and 2 for ABS-SUBTRACT

2.5.4. Saving the Inter-node Bindings

If, instead of just collecting the node names on a path, we pick up the variable bindings from node to node, then we will end up with a collection of path nodes together with a representation of the values of the variables therein. Once the path is collected in this way, the mapping between input variables and output variables can be derived by chaining along the variables on the path. The input elements are easily identified as the terminal nodes of the path. Actually, in principle any LHS pattern could be an input node. For now we assume that only patterns which do not depend on any RHS action are input elements. In Chapter 3, we will see that the program needs to be told which LHS patterns are input elements; in general, it is not safe to assume that only terminals are input elements.

By performing unifications as we search the network, we preserve a record of the dependencies between the variables. This dependency record can be traced from the output variables (i.e. those contained within a ***output** RHS action) all the way back to the input variables (i.e. those in the terminal nodes).

To circumvent the problem of variable name clashes, the variables are renamed with a numeric suffix representing their level in the graph (a 'level' is defined to be a whole rule (i.e. RHS, rule name, and LHS). Note that the numbers in figure 2-6 are *not* level numbers; they denote the order in which the nodes are visited during path collection). For example, the variables **?sign** and **?number** in the rule **halt1** are renamed to **?sign.1** and **?number.1** as they are at level 1. At the next level, in the rule subtract, the three variables become **?sign.2**, **?minuend.2** and **?subtrahend.2.**

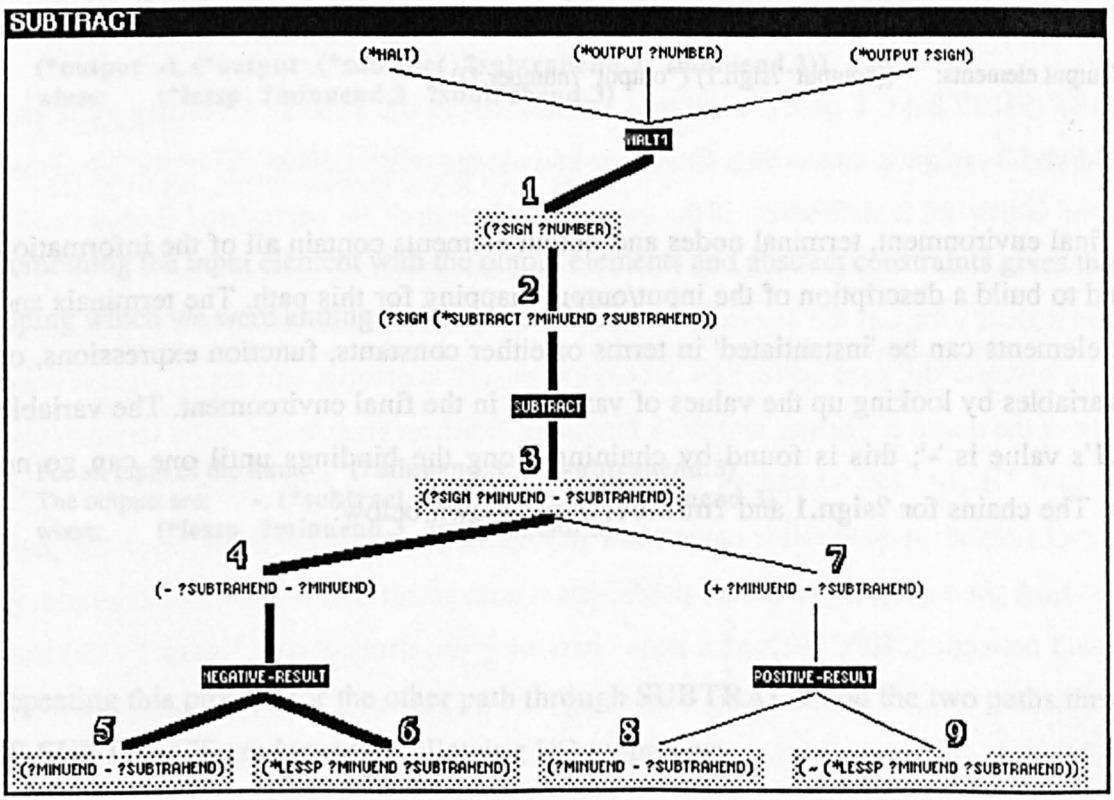


Figure 2-6 — First path through the SUBTRACT ruleset

For the first path through **SUBTRACT** (figure 2-6), augmenting the process of path collection in this way produces the following behaviour (where bindings between variables are held as variable/value pairs[1] in a structure called the 'environment'):

Unifying Node-1:	(?sign.1 ?number.1)
with Node-2:	(?sign.2 (*subtract ?minuend.2 ?subtrahend.2))
gives the environment:	(?sign.1/?sign.2 ?number.1/(*subtract ?minuend.2 ?subtrahend.2))
Unifying Node-3:	(?sign.2 ?minuend.2 - ?subtrahend.2)
with: Node-4	(- ?subtrahend.3 - ?minuend.3)
gives the environment:	(?sign.2/- ?minuend.2/?subtrahend.3 ?subtrahend.2/?minuend.3 ?sign.1/?sign.2 ?number.1/(*subtract ?minuend.2 ?subtrahend.2))

The output elements and terminal nodes, encountered on the way, are as follows:

Terminals:	((?minuend.3 - ?subtrahend.3) (*lessp ?minuend.3 ?subtrahend.3)
Output elements:	((*output ?sign.1) (*output ?number.1))

The final environment, terminal nodes and output elements contain all of the information required to build a description of the input/output mapping for this path. The terminals and output elements can be 'instantiated' in terms of either constants, function expressions, or input variables by looking up the values of variables in the final environment. The variable **?sign.1**'s value is '-'; this is found by chaining along the bindings until one can go no further. The chains for **?sign.1** and **?number.1** are shown below:

[1]Throughout this thesis, bindings are represented as the variable name, followed by the slash character, and then the variable's value. For example, the binding of ?x to John is represented as: ?x/John.

?sign.1 is bound to **?sign.2**
?sign.2 is bound to '-'

?number.1 is bound to **(*subtract ?minuend.2 ?subtrahend.2)**
?minuend.2 is bound to **?subtrahend.3**
?subtrahend.2 is bound to **?minuend.3**
?minuend.3 is unbound
?subtrahend.3 is unbound
thus, **?number.1**
becomes instantiated to **(*subtract ?subtrahend.3 ?minuend.3)**

The first terminal node, **(?minuend.3 - ?subtrahend.3)**, is the Working Memory element required to set the ruleset on its way. The **(*lessp ?minuend.3 ?subtrahend.3)** is a constraint on the values which **?minuend.3** and **?subtrahend.3** can take, and requires that **?minuend.3** be less than **?subtrahend.3**. If we use the final environment to instantiate the output elements, we get the following:

(*output -), (*output (*subtract ?subtrahend.3 ?minuend.3))
where: **(*lessp ?minuend.3 ?subtrahend.3)**

Combining the input element with the output elements and abstract constraints gives the I/O mapping which we were aiming for in section 2.4.2:

For an input of the form: **(?minuend.3 - ?subtrahend.3)**
The outputs are: **-, (*subtract ?subtrahend.3 ?minuend.3)**
where: **(*lessp ?minuend.3 ?subtrahend.3)**

Repeating this process for the other path through **SUBTRACT** and the two paths through **ABS-SUBTRACT** produces the following I/O mappings:

SUBTRACT path1:
For an input of the form: **(?minuend.3 - ?subtrahend.3)**
The outputs are: **-, (*subtract ?subtrahend.3 ?minuend.3)**
where: **(*lessp ?minuend.3 ?subtrahend.3)**

SUBTRACT path2:
For an input of the form: **(?minuend.3 - ?subtrahend.3)**
The outputs are: **+, (*subtract ?minuend.3 ?subtrahend.3)**
where: **(~ (*lessp ?minuend.3 ?subtrahend.3))**

ABS-SUBTRACT path1:
For an input of the form: **(?minuend.2 - ?subtrahend.2)**
The outputs are: **+, (*subtract ?minuend.2 ?subtrahend.2)**
where: **(~ (*lessp ?minuend.2 ?subtrahend.2))**

ABS-SUBTRACT path2:
For an input of the form: **(?minuend.2 - ?subtrahend.2)**
The outputs are: **+, (*subtract ?subtrahend.2 ?minuend.2)**
where: **(*lessp ?minuend.2 ?subtrahend.2)**
(~ (*lessp ?subtrahend.2 ?minuend.2))

2.5.5. The Need for a Semantics of Match Predicates

For **ABS-SUBTRACT** path2, the input constraints on the values of **?minuend.2** and **?subtrahend.2** are quite interesting. From our knowledge of arithmetic, it is clear to us that the second constraint is redundant; if the minuend is less than the subtrahend (constraint 1) then it follows that the subtrahend is not less than the minuend (constraint 2). The first constraint logically implies the second, so why does the system bother to include constraint-2? It does so because the path collection process is purely syntactic, and totally ignores the semantics of the match predicates and RHS functions. It has no knowledge of the semantics of the function ***lessp**, and so cannot see that **(*lessp ?a ?b)** $\supset$ **(~ (*lessp ?b ?a))**. The inclusion of redundant constraints causes few problems, but there are times when the path collector must reason about the match predicates it encounters on the way. For example, if during path collection, RPC derived a state containing the constraints: **(*lessp ?a ?b)** and **(*lessp ?b ?a)**, then it should stop searching down that route, as the constraints are *unsatisfiable*; it can never be the case that **?a** is less than **?b**, and **?b** is less than **?a**. In practice, such unsatisfiable paths could never be followed by a production system, because an assignment to the variables, satisfying the set of constraints, can never exist.

One reaction to this might be to counter that a production system should never define unsatisfiable paths - what is the point of defining a solution which can never be followed? In fact, production rule models are full of unsatisfiable paths. In general, constraints are added to

the LHS of a rule, to make sure that under certain conditions (i.e. those excluded by the constraints), the model will never go down that path. This is not easy to see in the abstract, so let us consider the following simple ruleset. For a given input of the form: **(first-feature ?x)**, it is meant to output two features of **?x**. If **?x** is an even number then, it will output **(?x is even), (and its successor is odd)**. However, if **?x** is odd, then we want it to fire rule **r2**, and then **r4**. The constraint ***oddp**, of rule **r4**, ensures that the ruleset never outputs: **(?x is even), (and its successor is even)**. In other words, the constraints ***evenp** and ***oddp** ensure that the path **r1** → **r4** is never followed.

```
(define-rule r1
 if (first-feature ?x) & (*evenp ?x)
 then (next-feature ?x) & (*output (?x is even)))

(define-rule r2
 if (first-feature ?x) & (*oddp ?x)
 then (next-feature ?x) & (*output (?x is odd)))

(define-rule r3
 if (next-feature ?x) & (*evenp ?x)
 then (*output (and its successor is odd)) & (*halt))

(define-rule r4
 if (next-feature ?x) & (*oddp ?x)
 then (*output (and its successor is even)) & (*halt))
```

If we run RPC on the above ruleset, it will find the following I/O mappings (the network is shown in figure 2-7):

Path1:
For an input of the form: **(FIRST-FEATURE ?X.2)**
The outputs is: **(AND ITS SUCCESSOR IS ODD)**
where: **(*EVENP ?X.2)**

Path2:
For an input of the form: **(FIRST-FEATURE ?X.2)**
The outputs is: **(AND ITS SUCCESSOR IS ODD)**
where: **(*ODDP ?X.2)**
(*EVENP ?X.2)

Path3:
For an input of the form: **(FIRST-FEATURE ?X.2)**
The outputs is: **(AND ITS SUCCESSOR IS EVEN)**
where: **(*EVENP ?X.2)**
(*ODDP ?X.2)

Path4:
For an input of the form: **(FIRST-FEATURE ?X.2)**
The outputs is: **(AND ITS SUCCESSOR IS EVEN)**
where: **(*ODDP ?X.2)**

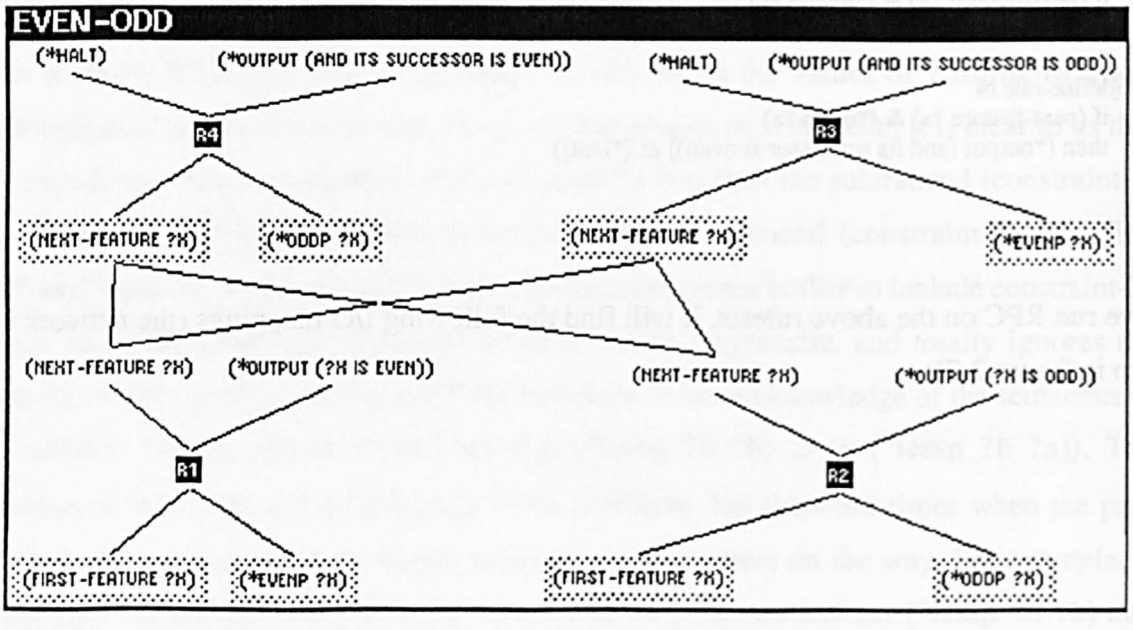


Figure 2-7 — Dependency network for rules r1, r2, r3, r4

The second and third paths are clearly unsatisfiable. There is no number which is both even and odd. Therefore, the ability to reason about the semantics of the match predicates within rules, is a prerequisite for deriving sensible I/O mappings. This is true for any program analyser, not just RPC; one can only see that path3 is contradictory if one knows (or can prove) that the predicates ***oddp** and ***evenp** are mutually exclusive. Any CP generator

which cannot prove that: $\neg\exists$(x) **NUMBER(x)** $\wedge$ **EVEN(x)** $\wedge$ **ODD(x)**[1], is doomed to an endless search for a number which is both even and odd.

2.5.6. Summary

The success of the RPC algorithm on the **ABS-SUBTRACT** and **SUBTRACT** models suggests that the approach has considerable promise. To deal with models containing contradictory paths, it needs to be augmented with a theorem prover, and an axiomatisation of the domain of interest. This is true of *any* program intended to reason about the I/O behaviour of production rule models. However, even with this augmentation, the algorithm is only suited to very restricted classes of models. RPC's limitations are the subject of the next section.

[1] A glossary of the symbols used in this thesis can be found in Appendix IIa; Appendix IIb contains a glossary of new terms introduced in this thesis.

2.6. Limitations of the RPC Algorithm

There are three major limitations of the RPC algorithm. For one, it cannot handle rulesets containing loops. In program analysis, interpreting the abstract behaviour of loops is a non-trivial problem. Another major limitation is RPC's inability to handle rules which contain more than one RHS pattern (not including function expressions). RPC can only handle rules which are Horn Clauses. Horn Clauses are rules where the righthand side of the implication contains only one term; there are no conjuncts. Ignoring function expressions such as **(*output ...)** because they do not deposit anything in working memory, the rules which we have encountered so far have all been Horn Clauses. However, in general we cannot bank on being so lucky.

It may be possible to extend the algorithm to handle looping rulesets containing non-Horn Clauses, but the final limitation is more damning and means that we will have to abandon reverse path collection altogether. Conflict resolution is a crucial component of production systems, yet RPC ignores it totally. Normally, this is fine as long as the rules do not interact (i.e. the conflict set only ever contains one element). This is why RPC was successful with the rulesets **SUBTRACT** and **ABS-SUBTRACT**. In these rulesets, the conditions of the rulesets are mutually exclusive - they are never instantiated at the same time. Furthermore, RPC relies on there being a neat flow of data from rule to rule. We will now deal with this criticism in more detail, before going on to propose a new method of deriving I/O mappings.

In the following ruleset, there is no simple flow of data from rule to rule. When the ruleset is run with the initial WME **(go)**, the rule **start** fires, depositing **(counter is 1)** in Working Memory. At this point, the rules **side-branch** and **halt** can both fire, but **side-branch** is chosen from the conflict set because it is more specific than **halt**. It outputs the number **1**, at which point **halt** fires because it is the only rule left in the conflict set. **Halt** outputs the number **2** and execution comes to a close.

```
(define-rule start
  if (go)
  then (counter is 1))

(define-rule side-branch
  if (counter is 1)
  then (*output 1))

(define-rule halt
  if (counter is ?x)
  then (*output (*add1 ?x)) & (*halt))
```

The PS Dependency Network, in figure 2-8, shows that, when analysed in terms of LHS/RHS dependencies, the rule **side-branch** is not on a direct line between **halt** and **start**. Searching backwards from the halt rule, RPC would miss the fact that the ruleset outputs the number **1**. Clearly, the analysis in terms of LHS/RHS dependencies is missing something. The problem is that the execution paths through a production system are not just defined by LHS/RHS dependencies. The possible execution paths are defined by the initial WMEs, the LHS/RHS dependencies, and *implicitly* by the process of conflict resolution. Any analysis of production rule models, which ignores conflict resolution, is doomed to failure.

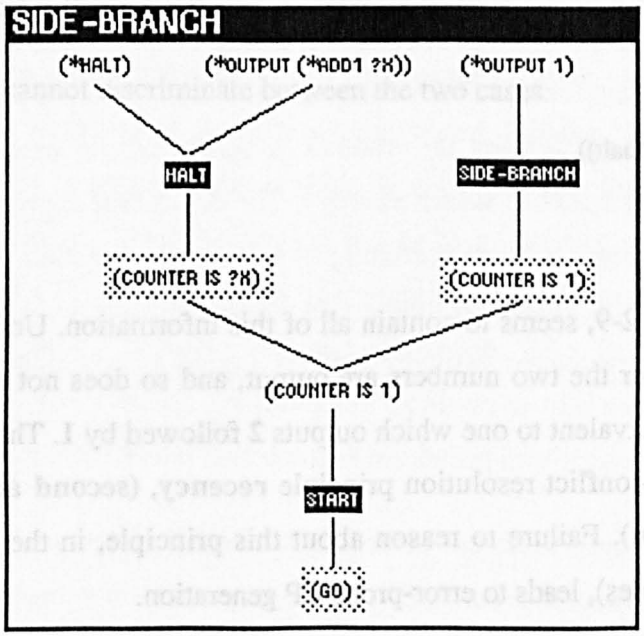


Figure 2-8 — Network demonstrating the need for conflict resolution

After inspecting the above network, the reader may feel that all is not lost - one could, perhaps, salvage the situation by searching *forwards* from the rule **start**. Upon reaching the choice point between the **side-branch** path and the **halt** path, the path collector could apply conflict resolution and decide that **(counter is 1)** is more specific than **(counter is ?x)**, and so search that path first, before returning to continue up the path towards **halt**. This may work for this simple, single LHS-pattern case, but it is not sufficiently general to cover cases where there are several LHS patterns, and the **recency** strategy plays a part in resolving the conflict.

The next example represents the final nail in the path collector's coffin. This trivial (but injurious) ruleset fires rule **r1** when a pattern of the form **(start)** is in Working Memory, depositing the patterns **(first action)** and **(second action)**. The rule **r2** outputs **1** if **(second action)** is present in Working Memory; **r3** is triggered by **(first action)**, outputs **2** and halts execution

```
(define-rule r1
  if (start)
  then (first action) & (second action))

(define-rule r2
  if (second action)
  then (*output 1))

(define-rule r3
  if (first action)
  then (*output 2) & (*halt))
```

The network in figure 2-9, seems to contain all of this information. Unfortunately, it does not tell us in which order the two numbers are output, and so does not reveal whether the model's behaviour is equivalent to one which outputs **2** followed by **1**. The rule **r2** only wins over **r3** because of the conflict resolution principle **recency**, **(second action)** being more recent than **(first action)**. Failure to reason about this principle, in the abstract (i.e. with WMEs containing variables), leads to error-prone CP generation.

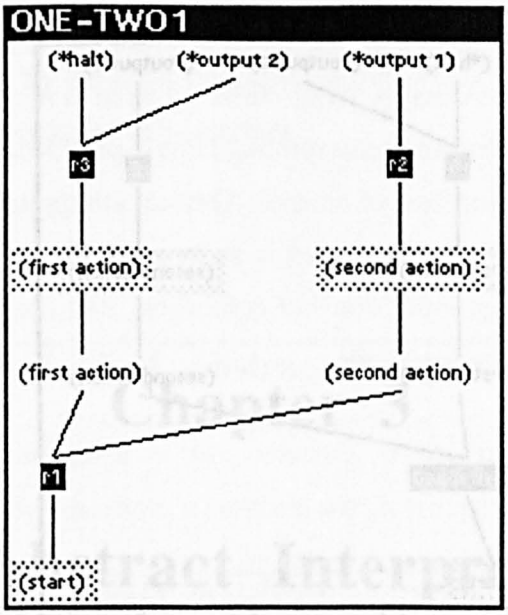


Figure 2-9 — Network where RECENCY is crucial

If we swap the righthand side actions of the rule **r1**, as shown below, then the network (figure 2-10) is identical (apart from the different name for **r1**).

```
(define-rule r1-swapped
  if (start)
  then (second action) & (first action))
```

It follows that RPC cannot discriminate between the two cases.

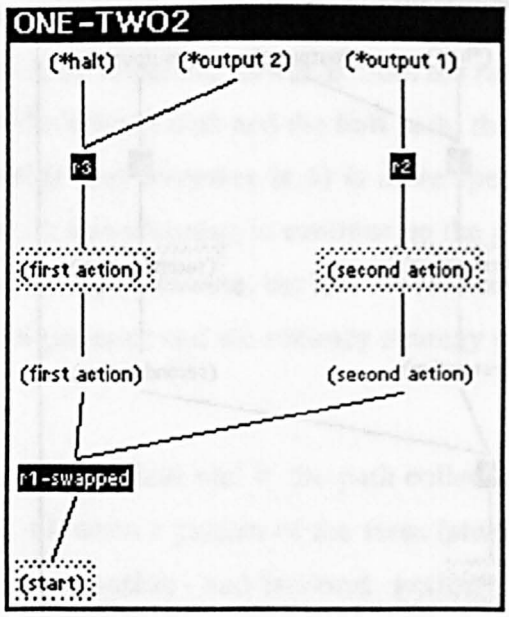


Figure 2-10 — Network where RECENCY is crucial

2.7. Conclusions

Despite its initial promise, reverse path collection based on LHS/RHS dependencies is not sufficiently powerful to capture all of the computational behaviour of production systems. Because conflict resolution defines the control structure of the production system, it determines the production system's search strategy. The depth-first approach of RPC totally ignores this control information. Even searching in a forwards direction does not overcome the basic need to reason about conflict resolution. In the next chapter, we develop an algorithm which can reason about how conflict resolution affects a given ruleset, in the abstract.

Chapter 3

The Abstract Interpretation of Production Systems

3.1. Introduction

In the previous chapter, we noted that scientists formulate experimental hypotheses in terms of variables which are either manipulable or measurable. These variables summarise the internal, unobservable workings of the system being tested. It is the descriptive economy of a theory, expressed solely in terms of observables, which simplifies the problem of proposing an informative experiment. This observation led us to propose a similar approach to the problem of automated CP generation, that of compiling the student model into an abstract description of the mapping between inputs and outputs. The Reverse Path Collection algorithm, designed to yield the I/O mappings of production rule models, was found to be seriously flawed because it could not take conflict resolution into account.

In this chapter we develop a different approach to the derivation of I/O mappings - an approach which overcomes the failings of RPC. This new approach is to run the student models *abstractly* (a process termed 'Abstract Interpretation'). The two competing models are run in tandem, and in the abstract (i.e. with uninstantiated variables). Given a pair of production rule models and a description of the class of problems which the student must solve, PG generates an abstract specification of the I/O behaviour of the two models.

In the interests of clarity, the algorithm for paired abstract interpretation of production rule models will be unveilled incrementally. After outlining what is required of the program, we will present an algorithm for abstractly interpreting simple production rule models, which do not rely on conflict resolution for flow of control. Abstract Interpretation requires the use of a theorem prover; theorem proving is reviewed in section 3.6. In sections 3.7 and 3.8, PG is enhanced so that it can reason about conflict resolution, and abstractly interpret pairs of rulesets in tandem, rather than one at a time.

3.2. Desiderata for a Critical Problem Generator

Our goal is to specify a procedure which takes a pair of production rule models, and generates a concrete input which discriminates between the two models, i.e. one for which each model produces a different string of outputs. On the way to generating a concrete input, it computes the input/output mappings of each ruleset. However, it is not necessary that PG find *all* I/O mappings for the two rulesets. It is sufficient to find just one from which a CP can be generated.

Although we do not require that PG find all I/O mappings, we do desire that its search strategy be *complete*. That is, in the limit, it should be able to find all I/O mappings, stopping when either a discriminating output description is found, or there are no more I/O mappings for the two rulesets. Furthermore, we require that the I/O mappings be *sound*. In other words, PG should never derive *abstract* I/O mappings encompassing *concrete* I/O mappings which the rulesets would never actually compute. For example, if a ruleset only computes the I/O mapping: (x + y) for the two input variables x, and y, then we would not expect PG to analyse the ruleset and derive the I/O mapping: (x - y). PG's derived I/O mapping would be *unsound* because it would embrace concrete I/O mappings which are never computed by the ruleset (e.g if x were 5 and y were 2, then PG would erroneously predict an output of 3). More formally, for any ruleset, if A is the set of possible I/O mappings in the concrete domain, and B is the set of possible instantiations (in the same concrete domain) of PG's derived I/O mappings, then it should never be the case that: $\neg B \subseteq A$.

In this thesis, we shall provide evidence that these principles of 'soundness' and 'completeness' are satisfied by the algorithm upon which PG is based, but we have not had time to develop a rigourous proof of this assertion.

3.3. Properties Required of Abstract Interpretation

Computer programs are designed to take *concrete* values as inputs, and produce other concrete values as outputs. For example, a program which computes the factorial of a number, would be given a concrete number, N, as input, and would output another number, N!. Now, imagine that our goal is to characterise the general behaviour of that program. We do not want to know what its output would be for some specific, concrete input; rather, we want some general description of the program's output for some abstract, non-specific N. The need to derive an abstract description of a program's behaviour, arises in a number of contexts. One such field is the automated debugging of computer programs. For example, one solution to the problem of debugging a novice programmer's code, is to first analyse the abstract computational behaviour of the program, and then compare that description with a library of pre-stored programming plans (Laubsch and Eisenstadt, 1981).

The process of abstractly executing a programming language has been variously termed 'Symbolic Evaluation' (Boyer/Moore), 'Meta-evaluation' (Wertz, 1982), 'Partial Evaluation' (Venken, 1984) and 'Abstract Interpretation' (Cousot and Cousot, 1977). In this thesis, we shall adopt the term 'Abstract Interpretation', as it seems better suited to production systems, where one talks of a Production System *Interpreter* rather than *Evaluator* (as in 'Lisp evaluator').

Abstract Interpretation is a commonly employed solution to the problems of automatic program analysis. The general idea is to glean information about a program by running it on *abstract specifications* of data objects, rather than the data objects themselves. Cousot and Cousot (1977) first introduced the notion of the Abstract Interpretation of imperative languages, proposing a 'static' semantics which associates each program statement with the set of possible machine states which might exist at that point. Mycroft (1981) has since developed this idea in the domain of functional programming languages. There is considerable interest in Abstract Interpretation amongst members of the Logic Programming community;

one reason for this interest is its usefulness in the design of optimising Prolog compilers (cf. Mellish (1986)).

3.4. The Concrete Operations of PGPS

In this section, we describe the various functional components of the PGPS interpreter. This analysis will form the basis for the design of PG. Our goal is to design PG so that it can *mimic*, in the abstract domain, the behaviour of PGPS in the concrete domain. By this we mean that PG's computational behaviour in the abstract domain should allow one to predict PGPS's behaviour in the concrete domain. In particular, we are interested in the I/O behaviour of production rule models.

The description of PGPS in the following section is an implementation-independent, but nevertheless fairly detailed specification of the interpreter. It has been included to provide a foundation upon which the specification of PG can be built. This should help clarify the difference between normal interpretation and Abstract Interpretation. Some aspects of the interpreter are described in both English and a mathematical notation. The mathematical definitions will not be used as part of any proofs, but have been included for the benefit of readers who prefer this level of precision.

Breaking down a production system interpreter into functional components is a somewhat arbitrary process. With this proviso in mind, we choose to conceive of the PGPS interpreter as consisting of four major functional sub-systems: Rule Instantiation, Conflict Resolution, Rule Firing, and the Recognise-act Cycle.

Rule Instantiation is the process of finding all of the rules whose lefthand sides are satisfied by the contents of Working Memory, and collecting those rules into the Conflict Set. Conflict Resolution involves selecting a unique instantiation from the members of the Conflict Set. Rule Firing entails carrying out the actions in the righthand side of the chosen instantiation, and the Recognise-act Cycle is responsible for passing control from one process to the next (Rule Instantiation → Conflict Resolution → Rule Firing → and back to the beginning again). We shall now describe these four processes in more detail.

3.4.1. Concrete Rule Instantiation

To compute the Conflict Set, the interpreter iterates over all of the rules in the ruleset, and collects instantiations of all of the rules which *match* Working Memory. Note that, if a rule matches more than one subset of the items in Working Memory, then the Conflict Set can contain more than one instantiation of that rule. Thus, for each rule, the interpreter obtains a (possibly empty) *set* of instantiations. The Conflict Set is merely the union of these sets (the instantiations of a given rule can be viewed as a 'set' rather than a 'bag' because Working Memory is not allowed to contain duplicate elements. It follows that there can never be two *equal* instantiations of a rule).

Definition of a Conflict Set:

Let $\{r_1,...,r_n\}$ be the set of rules in the ruleset,
and I_m denote the set of instantiations of rule r_m, where $1 \leq m \leq n$.
The Conflict Set is defined to be: $\{I_1 \cup ... \cup I_n\}$.

The instantiations of a rule are loosely defined to be the set of all possible ways of matching a rule's LHS with Working Memory. We will now refine this notion.

A PGPS LHS consists of a set of LHS-patterns (call it: **P**), a set of negated patterns, **N**, and a set of constraints, **C**, which define the relationships which must hold between variables in the LHS patterns. Each pattern in **P** can be paired with the members of **WM** which it matches, together with the bindings produced by the pattern matching process. We will call these triples 'Pattern Instantiations'. Matching an element of **P** with the elements of **WM** yields a set of Pattern Instantiations, **PI**. By a similar process, we can obtain the set of 'Negated Pattern Instantiations' for the members of **N** matched with the members of **WM**. A preliminary set of 'Rule Instantiations' can be formed from **PI** by computing the Cartesian Product of the sets therein. Some of these Rule Instantiations will be invalid, because they contain inconsistent variable bindings, violate one or more constraints in **C**, or violate one of the negated patterns of the rule. These invalid instantiations must be filtered out; the result of this filtration is the set of valid Rule Instantiations.

We now present a more formal definition of Pattern Instantiation Set and Preliminary Rule Instantiation Set.

Definition of a Pattern Instantiation Set:

If LHS is the set of conditions of some rule, r,
$P = \{x | x \in LHS \wedge Pattern(x)\}$,
and WM is the set of elements in Working Memory,
Then
$\forall(pattern \in P)$
Pattern-Instantiation-Set(r,{(pattern,wme,bindings)|wme∈WM ∧ Matches(pattern,wme,(),bindings)})[1]

Definition of a Preliminary Rule Instantiation Set:

For a rule, r, let PI = {x|Pattern-Instantiation-Set(r,x)}.
If $s_1,...s_n$ are the elements of PI,
Then the Preliminary Rule Instantiation Set is defined to be: $\{r\} \times s_1 \times ... \times s_n$.

The above definitions specify how a preliminary set of Rule Instantiations is derived. We will illustrate this with a fairly abstract example, which ignores the inner details of the patterns and WMEs. Each pattern and WME is represented by a subscripted alphabetic character. A pattern is said to match a WME if, ignoring the subscript, they are the same letter.

Let Working Memory, WM, be the set of six elements: $\{A_1,B_1,B_2,C_1,C_2,C_3\}$,
and let the rule, R, have the following LHS patterns: $\{B_3,A_2,C_4\}$.
The Pattern-Instantiation-Set of B_3 is:
$\{(B_3,B_1,b_1),(B_3,B_2,b_2)\}$, where b_n represents the bindings.
The Pattern-Instantiation-Set of A_2 is: $\{(A_2,A_1,b_3)\}$,
and that of C_4 is: $\{(C_4,C_1,b_4),(C_4,C_2,b_5),(C_4,C_3,b_6)\}$.
Thus, the Preliminary Rule Instantiation Set is:
$\{R\} \times \{(B_3,B_1,b_1),(B_3,B_2,b_2)\} \times \{(A_2,A_1,b_3)\} \times \{(C_4,C_1,b_4),(C_4,C_2,b_5),(C_4,C_3,b_6)\}$,
which is: { $(R,(B_3,B_1,b_1),(A_2,A_1,b_3),(C_4,C_1,b_4))$,
$(R,(B_3,B_1,b_1),(A_2,A_1,b_3),(C_4,C_2,b_5))$,
$(R,(B_3,B_1,b_1),(A_2,A_1,b_3),(C_4,C_3,b_6))$
$(R,(B_3,B_2,b_2),(A_2,A_1,b_3),(C_4,C_1,b_4))$,
$(R,(B_3,B_2,b_2),(A_2,A_1,b_3),(C_4,C_2,b_5))$,
$(R,(B_3,B_2,b_2),(A_2,A_1,b_3),(C_4,C_3,b_6))$ }

[1]The definition of **Matches** will be left until section 3.5.3, when it will be compared with PG's unification algorithm.

In general, some of the preliminary rule instantiations will be invalid, so we now need to set out the defining characteristics of such invalid instantiations, so that we know when to reject them. In PGPS there are four criteria for rejecting preliminary instantiations. The first restriction is that each pattern in the LHS must match a *unique* WME. Thus, in the above example, if the LHS pattern were $\{B_3,B_4,C_4\}$ instead of $\{B_3,A_2,C_4\}$, then we would not allow *both* B_3 and B_4 to match B_1, i.e. the preliminary instantiation $(R,(B_3,B_1,b_1),(B_4,B_1,b_2),(C_4,C_1,b_3))$ is invalid. We call this criterion 'WME-uniqueness':

WME-uniqueness:

Let the set of preliminary rule instantiations, PRI, be:
$\{I_1,...,I_n\}$ where $I_i=(r,(p_{i1},d_{i1},b_{i1}),...,(p_{im},d_{im},b_{im}))$,
$\forall(I_i \in PRI)\ d_j=d_k \wedge j\neq k \supset Invalid(I_i)$.

The second instantiation-rejection rule states that an instantiation is invalid if the bindings of the pattern/wme/binding triples are inconsistent with one another. A pattern/wme/binding triple is inconsistent with another if they both contain the same variable, and the variable's binding in the former triple is not equal to that in the latter. More formally:

Inter-pattern Consistency:

Let the set of preliminary rule instantiations, PRI, be:
$\{I_1,...,I_n\}$ where $I_i=(r,(p_{i1},d_{i1},b_{i1}),...,(p_{im},d_{im},b_{im}))$,
$\forall(I_i \in PRI)\ Inconsistent(b_{ij},b_{ik}) \supset Invalid(I_i)$.

Let the pair: (v,w) denote the binding of variable, v, to w,
B_i is the set of bindings: $\{(v_{i1},w_{i1}),...,(v_{im},w_{im})\}$.
B_j is the set of bindings: $\{(v_{j1},w_{j1}),...,(v_{jn},w_{jn})\}$.
$\forall(b_i \in B_i, b_j \in B_j)\ v_i=v_j \wedge w_i\neq w_j \supset Inconsistent(b_i,b_j)$.

The third restriction on instantiations is that none of the LHS constraints must be violated (this restriction is termed 'Constraint Fulfilment'). A constraint specifies the range of values which a variable can take. The range can be specified in terms of other variables and constants, for example, $x \geq y$ and $x \neq 0$ are both constraints. A constraint is violated if it is false

in the current 'environment' (the 'environment' is the set of bindings). For example, the constraint x≥y is false if x is bound to 3 and y is bound to 4, because 3≥4 is false. Thus:

Constraint Fulfilment:

If LHS is the set of conditions of some rule, r,
$C = \{x | x \in LHS \wedge Constraint(x)\}$.
Let the set of preliminary rule instantiations of r be:
$PRI = \{I_1,...,I_n\}$ where $I_i=(r,(p_{i1},d_{i1},b_{i1}),...,(p_{im},d_{im},b_{im}))$.
$\forall(I_i \in PRI)\ B_i = \{b_{i1} \cup ... \cup b_{im}\}$.
$\forall(I_i \in PRI, c \in C)\ False(evaluate(c,B_i)) \supset Invalid(I_i)$
where the function 'evaluate' instantiates its first argument, in terms of the bindings, and evaluates the resulting expression.

All that remains now is to specify the final criterion for instantiation rejection: an instantiation is invalid if there are any conflicting Negated Pattern Instantiations for that rule.

Definition of a Negated Pattern Instantiation Set:

If LHS is the set of conditions of some rule, r,
$N = \{x | x \in LHS \wedge Negated\text{-}Pattern(x)\}$,
WM is the set of elements in Working Memory,
and 'inverse' is a function which removes the negation sign from the negated pattern,
Then
$\forall(np \in N)$
$Negated\text{-}Pattern\text{-}Instantiation\text{-}Set(r,\{(np,wme,bindings) | wme \in WM \wedge Matches(inverse(np),wme,(),bindings)\})$

As defined above, a Negated Pattern Instantiation Set is the set of all triples consisting of the negated pattern, the WME it matches, and the bindings produced by the matching process. A Preliminary Rule Instantiation is invalid if the bindings in any Negated Pattern Instantiation are a subset of those in the Preliminary Rule Instantiation. That is:

Negated Pattern Fulfilment:

For a rule, r, let NPI = {x|Negated-Pattern-Instantiation-Set(r,x)}.
Let the set of preliminary rule instantiations of r be:
PRI = $\{I_1,...,I_n\}$ where $I_i=(r,(p_{i1},d_{i1},b_{i1}),...,(p_{im},d_{im},b_{im}))$.
$\forall(I_i \in PRI)\ B_i = \{b_{i1} \cup ... \cup b_{im}\}$.
$\forall(NPI_j \in NPI)$ Negated-$B_j = \{b_{j1} \cup ... \cup b_{jn}\}$.
$\forall(B_i, \text{Negated-}B_j)$ Negated-$B_j \subseteq B_i \supset$ Invalid(I_i).

So, a Rule Instantiation is the set of Preliminary Rule Instantiations which are not invalid (i.e. $\{I \in PRI \mid \neg Invalid(I)\}$). This completes our specification of the Conflict Set. We will now describe PGPS's conflict resolution strategy.

3.4.2. Concrete Conflict Resolution

The conflict resolution principles are used to choose a unique instantiation to fire. In PGPS there are three principles: Refractoriness, Recency, and Specificity. These principles were described in some detail in section 2.3.2, so in this section we will just present their formal definitions.

Refractoriness:

Let P be the set of instantiations fired on previous cycles,
and let CS be the set of instantiations in the current Conflict Set.
Refractoriness is a function which removes the previous instantiations
from the current Conflict Set, i.e.
Refractoriness: CS $\rightarrow$ CS - P.

Recency:

Let CS be the set of instantiations in the current Conflict Set.
Recency is a function which removes every instantiation which is
less recent than some other, i.e.
Recency: $CS \rightarrow CS - \{x \in CS \mid y \in CS \wedge \text{More-recent}(y,x)\}$.

Let (WM,$\sqsupset$) be the strictly-ordered set of WMEs.
Let the Conflict Set be: $CS = \{I_1,...,I_n\}$ where $I_i=(r,(p_{i1},d_{i1},b_{i1}),...,(p_{im},d_{im},b_{im}))$.
$\forall(I_i \in CS)\ W_i = \{d_{i1},...,d_{im}\}$.
$\forall(I_j \in CS)\ W_j = \{d_{j1},...,d_{jn}\}$.
$\forall(W_i,W_j)\ \exists(x \in W_i)\ \forall(y \in W_j \wedge y \notin W_i)\ x \sqsupset y \supset \text{More-recent}(I_i,I_j)$.

Specificity:

Let I be the set of instantiations in the current Conflict Set, CS.
Specificity is a function which removes every instantiation which is
less specific than some other, i.e.
Specificity: $I \rightarrow I - \{x \in I \mid y \in I \wedge \text{More-specific}(y,x)\}$.

$\forall(I_i,I_j \in CS)\ \text{Matches-one-way-only}(\text{lhs}(I_i),\text{lhs}(I_j)) \supset \text{More-specific}(I_j,I_i)$.[1]
$\forall(I_i,I_j \in CS)\ \text{Has-more-constraints}(\text{lhs}(I_i),\text{lhs}(I_j)) \supset \text{More-specific}(I_i,I_j)$.
$\forall(I_i,I_j \in CS)\ \text{Has-more-negated-patterns}(\text{lhs}(I_i),\text{lhs}(I_j)) \supset \text{More-specific}(I_i,I_j)$.
$\forall(I_i,I_j \in CS)\ \text{Negated-patterns-match-one-way-only}(\text{lhs}(I_i),\text{lhs}(I_j)) \supset \text{More-specific}(I_j,I_i)$.

Let 'sort' be a function which orders the elements in each LHS
by the WME matched to. Remember, at this point
all instantiations match the same WM subset (otherwise they would not be equally recent).
$\forall(x,y,b,c)\ \text{Matches}(\text{sort}(x),\text{sort}(y),(),b) \wedge \neg\text{Matches}(\text{sort}(y),\text{sort}(x),(),c) \supset \text{Matches-one-way-only}(x,y)$.

[1] In this particular definition only, each of these five Predicate Calculus formulae implicitly includes the negation of all its predecessors' antecedents. The negation includes the case where the I_i and I_j are swapped. For example, the second formula in this series should have the following pair of extra propositions: $\neg$Matches-one-way-only($\text{lhs}(I_i),\text{lhs}(I_j)) \wedge \neg$Matches-one-way-only($\text{lhs}(I_j),\text{lhs}(I_i)$). These extra negated propositions have not been included, because it makes the formulae harder to read (especially the fifth one, which requires six extra conjuncts). Therefore, in reading a particular formula, bear in mind that all its predecessors (in this set of five) must be false for the current one to be true.

Let C_i be the set of constraints in instantiation I_i,[1]
and C_j the set of constraints in instantiation I_j.
$\forall(I_i,I_j \in CS)\ \#C_i > \#C_j \supset$ Has-more-constraints(I_i,I_j).

Let N_i be the set of negated patterns in instantiation I_i,
and N_j the set of negated-patterns in instantiation I_j.
$\forall(I_i,I_j \in CS)\ \#N_i > \#N_j \supset$ Has-more-negated-patterns(I_i,I_j).

Let P-LHS$_i$ be the set of permutations of the LHS$_i$ of instantiation I_i,
and P-LHS$_j$ the set of permutations of the LHS$_j$ of instantiation I_j.
$\forall(I_i,I_j \in CS)\ \exists(x_i \in \text{P-LHS}_i, x_j \in \text{P-LHS}_j)\ \forall(b,c)$ Matches$(x_i,x_j,0,b) \wedge \neg$Matches$(x_j,x_i,0,c)$
$\supset$ Negated-patterns-match-one-way-only(x_i,x_j).

3.4.3. Concrete Rule Firing

When an instantiation of a rule fires, each action in the righthand side is executed in sequence. Each RHS is instantiated in terms of the instantiation's bindings. The patterns are added to Working Memory. A ***OUTPUT** expression causes its argument to be output. The action ***DELETE** leads to the removal of the referenced WME, and ***HALT** brings execution to a close. When an item, i, is deposited, it is added to Working Memory in such a way that it is **More-recent** than any of the other members (i.e. $\forall(e \in WM)\ i \sqsupset e$). Furthermore, the actions in the RHS are executed from left to right, thus RHS items to the right are **More-recent** than those to the left.

[1] Note that the set of constraints includes more than just the explicit function expressions in the LHS. For example, the following LHS contains *three* and not one constraint as one might have expected.

(number1 ?x) & (number2 ?x) & (number3 ?x) & (< ?x 10)

Apart from the obvious '>' constraint, there are also implicit equality constraints between the three ?x's. In counting the number of constraints, PGPS treats such implicit constraints as if they were explicit, by counting them as the equivalent minimal number of explicit equality constraints. Thus, because equality is a transitive and symmetric relation, the LHS can be re-expressed by adding just two equality constraints:

(number1 ?x) & (number2 ?y) & (number3 ?z) & (< ?x 10) & (= ?x ?y) & (= ?y ?z)

Transitivity means that the constraint (= ?x ?z) is redundant. Symmetry allows PGPS to leave out superfluous constraints such as: (= ?y ?x).

‖3.4.4. The Recognise-act Cycle and the PGPS Machine

We can view the running program and the PGPS-relevant parts of the computer's store as a self-contained 'machine', and refer to the 'state' of that virtual machine. From this perspective, it makes sense to refer to the *state* of the running PGPS machine at any one time. Obvious components of the PGPS machine include Production Memory, Working Memory and the component which implements the Recognise-act Cycle. Were we able to monitor the internal workings of this virtual machine, we could make observations such as: 'The machine is about to fire a rule', or: 'It is currently binding the variable, **?x**, to the number **5**'. In this section, we outline those aspects of the machine's state which are carried from cycle to cycle. This information is included to provide a baseline for comparison with PG.

For our purposes, there are only two aspects of a machine's state which can change from cycle to cycle. Working Memory normally changes when a rule fires, and this data structure must be maintained between firings. The interpreter must also keep a record of the instantiations fired. This record need only contain the rule name of the instantiation, and a list of the WMEs matched. The WMEs must be tagged in such a way that they can be distinguished from later *equal* elements which may have taken their place (for example, by giving each one a unique name).

Working Memory and the instantiations fired are the only changing information which must be maintained between cycles. In this chapter, we will discover that Abstract Interpretation requires a great deal more book-keeping between cycles.

3.5. Abstract Interpretation of Simple Production Systems

In the previous section, we outlined the workings of the PGPS interpreter. In this section, we will describe the Abstract Interpretation of PGPS rulesets which contain no negated patterns and do not involve any conflict resolution, other than the application of Refractoriness (in other words, after applying Refractoriness, the Conflict Set will contain at most *one* instantiation). In subsequent sections, new features will be added to the algorithm, culminating in an actual implementation, described in section 3.8.

Before embarking on the development of PG, it is worth pausing for a moment to review our goals for the proposed program. PG should be able to take an arbitrary pair of production rule models, and derive a concrete problem which causes each model to produce a different output. In Chapter 2, we said that CP generation can be reduced to two sub-problems: (i) the search for a pair of I/O mappings which are non-equivalent (one I/O mapping from each model), and (ii) the instantiation of those mappings. We shall discuss solutions to the latter problem in section 4.8.3.

3.5.1. The Need for an Input Specification

The goal of our Abstract Interpreter is to collect the outputs of a model, together with its inputs. A prerequisite of this goal is the provision of a description of the class of inputs which the model can consider (termed a set of 'Input Specifications'). It *is* possible to abstractly interpret a ruleset without an Input Specification, but the search space becomes unmanageable. In principle, any ordered subset of the LHS patterns could form a 'meaningful' abstract input to a model (by 'meaningful input' we mean one which is capable of matching a non-empty subset of the LHS patterns). For example, according to this criterion, a ruleset with five distinct LHS patterns is capable of spawning 5! Input Specifications, i.e. 120. In fact, there is nothing to stop us including duplicate abstract input elements, in which case the space of Input Specifications is infinite. Therefore, we would like to pass our Abstract Interpreter the set of Input Specifications which the models are intended to work on.

In the context of a tutoring system, this is a perfectly reasonable request. The tutoring system will already have such a description. The set of Input Specifications is precisely the collection of problem templates from which the tutoring system generates problems to pose to the student. These problem templates describe the set of *legal* problems for the domain being tutored. For example, a problem template in the subtraction domain might be: 'One number minus another, where the first number is greater than or equal to the second'. However, we would not expect our tutoring system to have the following template: 'One number, M, minus another, S, where the units digit of M is less than that of S, and M≥S'. This is because the reference to the relationship between the units digits of M and S has nothing to do with the legality of the problem; it is there to test for a particular subskill, i.e. borrowing. The units test is an implicit embodiment of the expectation that the borrowing subskill can be faulty, and implies that it is worth setting problems which require a borrow, as these will reveal any flaws in the student's algorithm.

PG will only be given Input Specifications which encode information about problem legality. To do otherwise, would be to bias its search for CPs.

3.5.2. A Simple Version of PG

The reader will recall that PGPS need only maintain a record of Working Memory and the instantiations fired. To handle Conflict-Resolution-free rulesets, PG must carry four other pieces of information. The first of these is a 'level' marker. This is a number denoting the current tree-depth of the problem-solving process. Every time a rule fires, we move one level deeper into the tree. As with RPC, the level marker is used to rename variables in a production rule, so that they do not clash with variables of the same name, in other parts of the tree.

PG must also maintain a record of the variable bindings, from cycle to cycle (termed an 'environment'). Working Memory will contain terms with uninstantiated variables, but PG can partially instantiate them by retrieving their values from the environment. In addition to an environment of variable bindings, PG must also maintain the set of 'constraints' on the variables in the environment. For example, if a firing rule requires that ?x be less than ?y, then this constraint must be carried forward to subsequent cycles.

Finally, PG must record any outputs, as these will be used to build the I/O mappings of the ruleset. The role played by these various datastructures will become clearer as the algorithm is developed in subsequent sections.

We will begin by describing the process of 'augmented unification' which enables PG to perform the abstract counterparts of PGPS's concrete operations. PGPS's one-way pattern matching is not sufficiently general to handle abstract data objects. Unification enables PG to produce abstract instantiations of rules (i.e. instantiations based on abstract data in WM). In order to ensure that the bindings generated by the process of unification are consistent, PG will need some kind of theorem proving capability (section 3.6).

We will find that negated patterns are particularly problematic in the abstract domain. PG is able to instantiate rules which contain negated patterns by deriving the minimal conditions under which the negated patterns are satisfied. The rule is only instantiated if these minimal conditions do not contradict the constraints collected so far. Firing such an instantiation is fairly simple because the hard work was done during the rule instantiation phase. All PG needs to do is deposit the righthand side actions in WM, uninstantiated.

3.5.3. Augmented Unification

As for PGPS, we choose to view PG as comprising four major functional components: Rule Instantiation, Conflict Resolution, Rule Firing, and the Recognise-act Cycle. In fact, PG has much in common with PGPS, which is not surprising given that it is meant to abstractly mimic PGPS's behaviour.

In keeping with PGPS, PG's Conflict Set is defined to be the union of the Rule Instantiations. However, the definition of 'Pattern Instantiation Set' is a little different. The difference lies in the fact that it uses *unification* rather than one-way pattern matching to compute the pattern instantiations. As promised in section 3.4.1, we will now define these two pattern matching procedures. Rather than express pattern matching and unification in terms of Predicate Calculus formulae, we will adopt Prolog as our specification language. This is because Predicate Calculus can be an unwieldy method of expressing what is effectively an algorithm. In particular, Prolog's use of *negation as failure*, and *clause ordering* leads to a significant reduction in the number of terms required to express relationships.

Because Predicate Calculus formulae are purely declarative, it is necessary to explicitly negate propositions which must be false for the whole formula to be true (cf. the definition of Specificity in section 3.4.2). Furthermore, if we define a proposition by material implication (⊃), and need to express statements about what is true when the proposition is false, then we must *explicitly* define the negation of that proposition. In Prolog, the negation of a proposition is implicitly entailed by one's failure to prove it.

The following two definitions of 'match' illustrate the greater economy of Prolog for some specification tasks. In this thesis, we have adopted the Edinburgh syntax for Prolog (Clocksin and Mellish, 1981) as this is the generally accepted standard. The only difference is that we use a tilde ('~') to symbolise quasi-negation, rather than the Edinburgh '\+'[1], we use '<-' rather than ':-', and we have used '&' rather than a comma to separate conjuncts in a clause (the latter two changes should serve as a constant reminder to the reader that this is pseudo-code rather than executable Prolog). Note that in Prolog, variables begin with an uppercase letter (and are implicitly universally quantified), whilst constants and clause functors begin with a lower case character (in contrast to Predicate Calculus).

[1] Our quasi-negation operator can be defined as follows:

```
op(60,fx,~).
~X :- \+X.
```

Predicate Calculus definition of 'match':

$\forall(x,y,env)\ x=y \supset Matches(x,y,env,env).$
$\forall(x,y,env)\ Constant(x) \wedge Constant(y) \wedge x \neq y \supset \neg Matches(x,y,env,env).$
$\forall(x,y,env)\ Variable(x) \wedge Bound(x,env) \wedge lookup(x,env)=y \supset Matches(x,y,env,env).$
$\forall(x,y,env)\ Variable(x) \wedge Bound(x,env) \wedge lookup(x,env) \neq y \supset \neg Matches(x,y,env,env).$
$\forall(x,y,env,new\text{-}env)$
$Variable(x) \wedge \neg Bound(x,env) \wedge Bind(x,y,env,new\text{-}env) \supset Matches(x,y,env,new\text{-}env).$
$\forall(x,y,env,head\text{-}env,new\text{-}env)$
$List(x) \wedge List(y)$
$\wedge Matches(head(x),head(y),env,head\text{-}env) \wedge Matches(tail(x),tail(y),head\text{-}env,new\text{-}env)$
$\supset Matches(x,y,env,new\text{-}env).$
$\forall(x,y,env,head\text{-}env,new\text{-}env)$
$List(x) \wedge List(y)$
$\wedge (\neg Matches(head(x),head(y),env,head\text{-}env) \vee \neg Matches(tail(x),tail(y),head\text{-}env,new\text{-}env))$
$\supset \neg Matches(x,y,env,new\text{-}env).$

Pseudo-Prolog definition of 'match':

```
/* The fourth argument is an output variable. */
matches(X,X,Env,Env).
matches(X,Y,Env,Env) <-
  variable(X) & bound(X,Env) & lookup(X,Env,Y).
matches(X,Y,Env,NewEnv) <-
  variable(X) & ~bound(X,Env) & bind(X,Y,Env,NewEnv).
matches([X|Xtail],[Y|Ytail],Env,NewEnv) <-
  matches(X,Y,Env,HeadEnv) & matches(Xtail,Ytail,HeadEnv,NewEnv).
```

The following definition of 'unify' is augmented to handle the unification of function expressions with constants or other function expressions (this augmentation was also used in the derivation of PS Dependency Networks, as outlined in section 2.5.2). The relation **addEqualityConstraint** performs this function. For example, when asked to unify the function expression: **(*times ?x 5)** with: **20**, it generates the equality constraint: **(= (*times ?x 5) 20)**. The subsidiary relation, **lookup**, computes the 'Ultimate Associate' of **X** in **Env**. It completely instantiates lists, so if **X** is a list containing variables, then those variables are fully instantiated also. The relation, **bind**, contains an 'Occurs check', and fails if its first parameter *occurs* in its second. The exception to this rule is where the second parameter is a function expression; under these conditions it adds an equality constraint (but *not* a binding) between the first and second parameters. This refinement allows the unification algorithm to compute constraints such as: **(= ?x (*times ?x ?y))**. This equality constraint could be used by a suitably-equipped theorem prover to infer that **?y** must have the value: **1**.

Pseudo-Prolog definition of augmented 'unify':

```
/* The fourth and fifth arguments are output variables.*/
unify(X,X,Env,Constraints,Env,Constraints).
unify(X,Y,Env,Constraints,Env,Constraints) <-
  lookup(X,Env,ValX) & lookup(Y,Env,ValX).
unify(X,Y,Env,Constraints,NewEnv,NewConstraints) <-
  lookup(X,Env,ValX) & lookup(Y,Env,ValY) & variable(ValX) &
  bind(ValX,ValY,Env,Constraints,NewEnv,NewConstraints).
unify(X,Y,Env,Constraints,NewEnv,NewConstraints) <-
  lookup(X,Env,ValX) & lookup(Y,Env,ValY) & variable(ValY) &
  bind(ValY,ValX,Env,Constraints,NewEnv,NewConstraints).
unify(X,Y,Env,Constraints,Env,NewConstraints) <-
  lookup(X,Env,ValX) & lookup(Y,Env,ValY) & functionExpression(ValX) &
  addEqualityConstraint(ValX,ValY,Constraints,NewConstraints).
unify(X,Y,Env,Constraints,Env,NewConstraints) <-
  lookup(X,Env,ValX) & lookup(Y,Env,ValY) & functionExpression(ValY) &
  addEqualityConstraint(ValY,ValX,Constraints,NewConstraints).
unify(X,Y,Env,Constraints,NewEnv,NewConstraints) <-
  lookup(X,Env,ValX) & lookup(Y,Env,ValY) &
  ValX = [HeadValX|TailValX], ValY = [HeadValY|TailValY],
  unify(HeadValX,HeadValY,Env,Constraints,EnvForHead,ConstraintsForHead) &
  unify(TailValX,TailValY,EnvForHead,ConstraintsForHead,NewEnv,NewConstraints).
```

3.5.4. Deriving Abstract Rule Instantiations

When PG is run on an Input Specification, the level marker is initialised at zero, and the patterns in the Input Specification are added to Working Memory, after *renaming* the variables therein. Any constraints in the Input Specification are also renamed, and are then added to PG's constraint-set. For example, if the Input Specification were: **(?x - ?y), (> ?x ?y)**, then Working Memory would contain the single element: **(?x.0 - ?y.0)**, and the constraint-set would consist of: **(> ?x.0 ?y.0)**. PG is now ready to commence the first Recognise-act Cycle. The Recognise-act Cycle begins with the incrementing of the level marker to **1**.

We can now define the abstract versions of 'Pattern Instantiation Set' and 'Preliminary Rule Instantiation Set'. The major difference between the abstract and concrete versions lies in the former's use of unification rather than one-way pattern matching. Furthermore, the variables in the rules must be renamed using the current level, so that their bindings do not clash with those of earlier variables of the same name. The patterns of a rule are unified with the members of Working Memory, to yield a set of Abstract Pattern Instantiations, each consisting of the pattern, WME matched, the bindings produced, and any constraints generated during unification.

Definition of an Abstract Pattern Instantiation Set:

If LHS is the set of conditions of some rule, r,
env is the current environment,
P = {x|x∈LHS ∧ Pattern(x)},
and WM is the set of elements in Working Memory,
Then
∀(patterne∈ P)
Abstract-Pattern-Instantiation-Set
(r,{(pattern,wme,new-env,new-constraints)
|wme∈ WM ∧ Unify(pattern,wme,env,(),new-env,new-constraints)})

The definition of a Preliminary Abstract Rule Instantiation Set is almost the same as that of a concrete one. The difference lies in the fact that it is built from abstract rather than concrete Pattern Instantiation Sets. This means that it will contain the accumulated bindings, rather than just those produced during the pattern match.

Definition of a Preliminary Abstract Rule Instantiation Set:

For a rule, r, let API = {x|Abstract-Pattern-Instantiation-Set(r,x)}.
If $s_1,...s_n$ are the elements of API,
Then the Preliminary Rule Instantiation Set is defined to be: $\{r\} \times s_1 \times ... \times s_n$.

Of course, some of these preliminary instantiations may be invalid, as would be the case if the variables in the separate patterns were inconsistently bound with one another. The four instantiation-validity criteria in PGPS are: WME-uniqueness, Inter-pattern Consistency, Constraint Fulfilment, and Negated Pattern Fulfilment. We will now define their abstract counterparts.

Abstract WME-uniqueness:

Let the set of preliminary abstract rule instantiations, PARI, be:
$\{I_1,...,I_n\}$ where $I_i=(r,(p_{i1},d_{i1},b_{i1},c_{i1}),...,(p_{im},d_{im},b_{im},c_{im}))$,
$\forall(I_i \in PRI)\ d_j=d_k \wedge j\neq k \supset Invalid(I_i)$.

The second concrete instantiation-rejection rule requires that the inter-pattern variable bindings be consistent. Two bindings are said to be inconsistent if they are bound to different items. In the abstract case, two bindings are inconsistent if they cannot be unified with one another. Our

definition of Abstract Inter-pattern Consistency depends crucially on the notion of a 'merge'. Two Abstract Pattern Instantiations are said to be consistent if and only if it is possible to *merge* their respective environments. Consider the following pair of Abstract Pattern Instantiations.

API_1: Pattern=(?x.1 likes ?y.1)
WME=(?a.0 likes ?b.0)
Env=(?y.1/?b.0 ?x.1/?a.0)

API_2: Pattern=(?y.1 likes ?x.1)
WME=(?b.0 likes ?c.0)
Env=(?x.1/?c.0 ?y.1/?b.0)

The above pair is consistent because, although **?x.1** is bound to **?a.0** in API_1 and **?c.0** in API_2, the variables **?a.0** and **?c.0** are unbound and so can be unified with one another, yielding the merged environment: **(?a.0/?c.0 ?y.1/?b.0 ?x.1/?a.0)**. However, API_3 and API_4 are not consistent, because **?a.0** and **?c.0** are bound to unequal constants, and so cannot be unified.

API_3: Pattern=(?x.1 likes ?y.1)
WME=(?a.0 likes ?b.0)
Env=(?y.1/?b.0 ?x.1/?a.0 ?a.0/JOHN)

API_4: Pattern=(?y.1 likes ?x.1)
WME=(?b.0 likes ?c.0)
Env=(?x.1/?c.0 ?y.1/?b.0 ?c.0/MARY)

The environment-merging procedure produces the most general unifier of the compound pair of patterns unified with the compound pair of WMEs. In other words, it yields an environment which is identical to that obtained by unifying the pattern-pair with the WME-pair, *in one pass*. So, merging the environments of API_1 and API_2 is equivalent to performing the following unification:

Unify: ((?x.1 likes ?y.1) (?y.1 likes ?x.1))
with: ((?a.0 likes ?b.0) (?b.0 likes ?c.0))
gives: (?a.0/?c.0 ?y.1/?b.0 ?x.1/?a.0)

Merging environments, rather than repeating the whole unification, is worthwhile because it saves redundant computational effort. We will now define the 'merge' algorithm in pseudo-

Prolog. This algorithm also merges the constraints of the two patterns; constraints are merged by conjoining the two sets. Note that the process of unification can generate new equality constraints; these must be included in the returned constraint-set. Also, **Env2** is reversed so that the bindings are merged in the correct order.

Definition of 'merge':

```
/* Arguments five and six are output variables.
   The predicate CONJOIN produces the conjunction of two sets of constraints,
   e.g. conjoin([A],[B]) => [A,&,B]. */
merge(Env1,Env2,Constraints1,Constraints2,MergedEnv,MergedConstraints) <-
  reverse(Env2,ReversedEnv2) & /* Reverse to preserve order. */
  conjoin(Constraints1,Constraints2,ConjoinedConstraints) &
  merge1(ReversedEnv2,Env1,ConjoinedConstraints,MergedEnv,MergedConstraints).

merge1([],Env1,Constraints,Env1,Constraints).
merge1([Binding2|RestEnv2],Env1,ConjoinedConstraints,MergedEnv,MergedConstraints) <-
  varOf(Binding2,Var) &
  valOf(Binding2,ValueInEnv2) &
  lookup(Var,Env1,ValueInEnv1) &
  unify(ValueInEnv1,ValueInEnv2,[],ConjoinedConstraints,[NewBinding],NewConstraintSet) &
  merge1(RestEnv2,[NewBinding|Env1],NewConstraintSet,MergedEnv,MergedConstraints).
```

We are now in a position to define inter-pattern consistency for abstract patterns. In the concrete case, we were able to define this criterion in terms of a *pair* of Pattern Instantiations. In order to reject a Preliminary Rule Instantiation, it was sufficient to find a pair of patterns which were inconsistently bound. In the abstract case, this will not do. Unification of the pattern-set with Working Memory can lead to a chain of dependencies amongst the variables in *all* of the patterns. Therefore, finding a pair of patterns which cannot be merged, will not suffice as a rejection criterion. We cannot compare two patterns in isolation; it is necessary to ascertain whether the pattern set *as a whole* can be merged. Furthermore, the merged environment (and constraints) must be retained for use when checking that the abstract negated patterns are satisfied. The following pseudo-Prolog definition sets out how to compute the merged environment and constraints. A Preliminary Abstract Rule Instantiation is invalid if it is not possible to merge the environments and constraints.

Pseudo-Prolog Definition of Abstract Inter-pattern Consistency:

```
/* The predicate 'constraints' extracts the constraints of a rule.
   'globalConstraints' has access to the set of constraints collected so far.*/
abstractInterPatternConsistent([RulelPatternSet]) <-
  constraints(Rule,Constraints) &
  globalConstraints(GlobalConstraints) &
  conjoin(Constraints,GlobalConstraints,ConjoinedConstraints) &
  mergePatterns(PatternSet,[],ConjoinedConstraints,MergedEnv,MergedConstraints).

/* The final two arguments of this clause are output arguments.
   The predicate 'consistent' is a call to the theorem prover. A set of constraints is consistent
   if the theorem prover fails to find a contradiction. The theorem prover also simplifies the
   constraint-set by removing duplicate constraints, tautologies and subsumed clauses.
   These operations will described in the section on Theorem Proving. */
mergePatterns([],EnvMergedSoFar,ConstraintsMergedSoFar,EnvMergedSoFar,
                                                        SimplifiedConstraints) <-
  consistent(ConstraintsMergedSoFar,EnvMergedSoFar,SimplifiedConstraints).

mergePatterns([PlRestOfPatternSet],EnvMergedSoFar,ConstraintsMergedSoFar,FinalEnv,
                                                                  FinalConstraints) <-
  P=[Pattern,WME,Env,Constraints],
  merge(EnvMergedSoFar,Env,ConstraintsMergedSoFar,Constraints,NewEnv,NewConstraints) &
  mergePatterns(RestOfPatternSet,NewEnv,NewConstraints,FinalEnv,FinalConstraints).
```

In PGPS, none of the LHS constraints must be violated (this restriction was termed 'Constraint Fulfilment'). The principle of Constraint Fulfilment is incorporated in the **mergePatterns** definition. The patterns can only be merged if the constraints are *consistent* in the merged environment. Concrete Constraint Fulfilment was checked by 'evaluating' the function expression (constraint) in the current environment. In general, we cannot do this in an abstract domain, so the job must be done by a theorem prover which can assess consistency, in the abstract. Theorem proving is reviewed in section 3.6. Until then, we will assume that PG has access to an 'oracle' which can always answer the question: "Is this set of constraints consistent in the current environment?".

We must now define the abstract version of Negated Pattern Fulfilment. First of all we will define the term Abstract Negated Pattern Instantiation Set. This definition is similar to the concrete one, but uses **Unify** rather than **Matches**. Note that the set of bindings stored in the instantiation is the collection of new bindings (i.e. those in **bindings** which are not in **env**). We shall see momentarily that **bindings - env** is required for the definition of Abstract Negated Pattern Fulfilment; it is converted into a set of equality constraints which, when negated, form what we term a Minimal Negation Nullifier.

Definition of an Abstract Negated Pattern Instantiation Set:

If LHS is the set of conditions of some rule, r,
env is the current environment,
N = {x|x∈LHS ∧ Negated-Pattern(x)},
WM is the set of elements in Working Memory,
and 'inverse' is a function which removes the negation sign from the negated pattern,
Then
∀(np∈N)
Negated-Pattern-Instantiation-Set
(r,{(np,wme,bindings - env,constraints)|wme∈WM
∧ Unify(inverse(np),wme,env,(),bindings,constraints)})

In keeping with the concrete case, we require that none of the negated patterns of an instantiation be violated. In the concrete case, this reduces to the task of checking that none of the bindings of any Negated Pattern Instantiations are a subset of those of the Preliminary Rule Instantiation. For Abstract Interpretation, the decision is not so simple. The problem is that, because we are dealing with uninstantiated variables, it is not always clear whether a negated pattern is violated by the contents of Working Memory. The example below serves to illustrate this assertion. Working Memory contains three elements, the first one, **(?a ?b)**, is matched by: **(?x ?y)**. Elements 2 and 3 of Working Memory are matched by the negated pattern. We might be tempted to conclude that this rule cannot fire because the negated pattern matches something in Working Memory. Such a conclusion is flawed, because we do not know what the variables, **?a**, **?b**, **?c**, **?d** and **?e**, are bound to, and so cannot tell whether **?x** and **?y** in the positive pattern are bound to something which is equal to the bindings of **?x** and **?y** in the negated pattern.

LHS: (?x ?y) & ~(tried ?x ?y)
WM: ((?a ?b) (tried ?c ?d) (tried ?e ?b))

Bindings of first pattern: (?y/?b ?x/?a)
Bindings of negated instantiation1: (?y/?d ?x/?c)
Bindings of negated instantiation2: (?y/?b ?x/?e)

The only way that this rule can fire is if the positive bindings of **?x** and **?y** are not equal to those of the negated pattern. Let us work out the conditions under which this is true. The rule can fire provided that **(tried ?x ?y)** matches *neither* **(tried ?c ?d)** nor **(tried ?e ?b)**. For

the second WME, this would be true if **?x** could not be bound to **?c**, *or* **?y** could not bound to **?d**. For the third, **(tried ?x ?y)** fails to match if either **?x≠?e** or **?y≠?b**. Let us consider the latter case where: **(?x≠?e ∨ ?y≠?b)**. From the positive pattern we know that **?x** is bound to **?a**, and **?y** is bound to **?b**. Therefore, the disjunction: **(?x≠?e ∨ ?y≠?b)** is true if **(?a≠?e ∨ ?b≠?b)**. This reduces to **?a≠?e**, because **?b≠?b** is unsatisfiable. A similar chain of reasoning leads us to conclude that the negated pattern will not match the second WME, **(tried ?c ?d)**, if **(?a≠?c ∨ ?b≠?d)**. So, the rule can fire provided that both of these conditions are satisfied, i.e. **?a≠?e ∧ (?a≠?c ∨ ?b≠?d)**. We call this expression a 'Minimal Negation Nullifier', because it describes the *minimal* conditions for the negation to be annulled. The instantiation and removal of unsatisfiable expressions, such as **?b≠?b**, is performed by the theorem prover. The theorem prover is actually passed a 'Preliminary Negation Nullifier' - so named because it is uninstantiated and may contain unsatisfiable expressions.

To build a Preliminary Negation Nullifier, we take the bindings in each Abstract Negated Pattern Instantiation and turn them into equality constraints. The set of equality constraints from each Abstract Negated Pattern Instantiation is then negated, signifying that each set should be false. The conjunction of these negated sets is the Preliminary Negation Nullifier, as shown below.

Equality constraints from first instantiation: $((= ?y\ ?d) \wedge (= ?x\ ?c))$
Negated constraints: $\neg((= ?y\ ?d) \wedge (= ?x\ ?c))$
Equality constraints from second instantiation: $((= ?y\ ?b) \wedge (= ?x\ ?e))$
Negated constraints: $\neg((= ?y\ ?b) \wedge (= ?x\ ?e))$

Preliminary Negation Nullifier: $\neg((= ?y\ ?d) \wedge (= ?x\ ?c)) \wedge \neg((= ?y\ ?b) \wedge (= ?x\ ?e))$

The above Preliminary Negation Nullifier looks slightly different to that which we generated by informal argument, but is in fact equivalent. By *de Morgan's* Law, $\neg(x \wedge y) \equiv (\neg x \vee \neg y)$; therefore, we can rewrite the expression to the following.

$(\neg(= ?y\ ?d) \vee \neg(= ?x\ ?c)) \wedge (\neg(= ?y\ ?b) \vee \neg(= ?x\ ?e))$

If we then instantiate this expression in terms of the current environment, and simplify, we obtain the same expression as that derived by informal argument.

Instantiating: $(\neg(= ?y\ ?d) \vee \neg(= ?x\ ?c)) \wedge (\neg(= ?y\ ?b) \vee \neg(= ?x\ ?e))$
in the environment: (?y/?b ?x/?a)
gives: $(\neg(= ?b\ ?d) \vee \neg(= ?a\ ?c)) \wedge (\neg(= ?b\ ?b) \vee \neg(= ?a\ ?e))$

Removing unsatisfiable disjuncts gives: $(\neg(= ?b\ ?d) \vee \neg(= ?a\ ?c)) \wedge \neg(= ?a\ ?e)$
which is a syntactic variant of: $?a \neq ?e \wedge (?a \neq ?c \vee ?b \neq ?d)$

The Preliminary Negation Nullifier, together with the current environment and constraints, is passed to the theorem prover for consistency checking. If the Preliminary Negation Nullifier is unsatisfiable in the current environment, then the instantiation must be rejected. If it is satisfiable, then the theorem prover returns the Minimal Negation Nullifier. This constraint-set is added to the set of global constraints so that it can be used during subsequent Recognise-act Cycles.

The above notions are specified more formally, below. The following definition works on instantiations which have made it through the previous filters. It assumes that the Abstract Inter-pattern Consistency filter has been applied, which means that the bindings have been merged, as has the constraint-set. In order to keep things simple, the definition does not show the fact that the instantiation acquires the value of **ReturnedConstraints** as its new constraint-set.

Pseudo-Prolog Definition of Abstract Negated Pattern Fulfilment:

```
satisfiesNegations(I,ReturnedConstraints) <-
  instantiationEnvironment(I,MergedEnv) & /* Accesses the environment of the instantiation. */
  instantiationConstraints(I,MergedConstraints) & /* Accesses the constraints of the instantiation. */
  abstractNegatedPatternInstantiationSet(ANPIS) &
  buildPreliminaryNegationNullifier(ANPIS,PreliminaryNegationNullifier) &
  conjoin(PreliminaryNegationNullifier,MergedConstraints,ConjoinedConstraints) &
  consistent(ConjoinedConstraints,MergedEnv,ReturnedConstraints).


buildPreliminaryNegationNullifier([],[]).
buildPreliminaryNegationNullifier([ANPI|RestofAbstractNegatedPatternInstantiationSet],
                                  NullifyingConstraintSet) <-
  ANPI=[NegPattern,WME,Env,Constraints],
  turnBindingsIntoEqualityConstraints(Env,EqualityConstraints) &
  conjoin(EqualityConstraints,Constraints,ConjoinedConstraints) &
  negateConstraintSet(ConjoinedConstraints,NegatedConstraints) &
  buildPreliminaryNegationNullifier(RestofAbstractNegatedPatternInstantiationSet,
                                    OtherNegatedConstraints) &
  conjoin(NegatedConstraints,OtherNegatedConstraints,NullifyingConstraintSet).
```

So, every Preliminary Abstract Rule Instantiation, which makes it through all of these filters, is added to the Conflict Set. Such fully-fledged Abstract Rule Instantiations will also have had the separate Abstract Pattern Instantiation environments merged, as well as the constraints. Furthermore, any Minimal Negation Nullifiers will also have been merged with the constraint-set. The above definition does not specify how to negate an empty set of constraints (an empty set of constraints arises when an Abstract Negated Pattern Instantiation contains no bindings or unification constraints). The truth value of an empty set of constraints is 'T', because there are no constraints to satisfy. It follows that its negation is false ('F'), which is to say that there is no way that the negated pattern could *not* match the WME in question.

The next section describes how PG fires Abstract Rule Instantiations. Conflict Resolution will be described in section 3.7.

3.5.5. Abstract Rule Firing

Abstract Rule Firing is relatively trivial. Recall that the RHS actions have already been renamed with the current level marker. Each RHS action is executed in turn: the RHS patterns are deposited in Working Memory; any ***OUTPUT** expressions are added to the list of

outputs; ***DELETE** expressions result in the removal of the abstract WME; and ***HALT** arrests cycling.

Provided that a halt was not signalled, the Recognise-act Cycle recommences with the incrementing of the level marker. If the Conflict Set is empty after computing the Abstract Rule Instantiations and applying Abstract Refractoriness, then the interpreter halts.

We shall now consolidate the ideas introduced in earlier sections by working through an example which is designed to illustrate the processes described so far. The Abstract Interpretation of this ruleset will entail the generation of a unification constraint, and a Minimal Negation Nullifier. We will assume that Refractoriness works in the abstract, as it is not described until section 3.7.

3.5.6. A Simple Example of Abstract Interpretation

The following tiny ruleset has been created to illustrate how a negation nullifier is built, and to show the unification of a function expression with a constant. In the concrete domain, it starts with an input of the form: **(add ?x ?y)**, where the variables are both natural numbers, and deposits the sum of **?x** and **?y** in working memory. The rule **finish** fires if that sum is zero, not equal to **?x**, and the original **?x** and **?y** are equal; it outputs **?x** and halts. Clearly, there is no way that this ruleset can ever output anything, because the formula: **?x+?y=0** $\wedge$ **?x=?y** $\wedge$ **?x≠0** $\wedge$ **NATNUM(?x)** $\wedge$ **NATNUM(?y)** is unsatisfiable. In the following scenario, we shall see how the attempt to create a negation nullifier leads to the predicted contradiction.

Input Specification: ((add ?x ?y) (*natnum ?x) (*natnum ?y))

```
(define-rule start
  if (add ?x ?y)
  then (sum-x-y (*plus ?x ?y)))

(define-rule finish
  if (add ?x ?x) & (sum-x-y 0) & ~(sum-x-y ?x)
  then (*output ?x) & (*halt))
```

At the start of the run, Working Memory is initialised with the single element: **(add ?x.0 ?y.0)**, and the constraint-set is initialised to: **(*natnum ?x.0)** $\wedge$ **(*natnum ?y.0)**. The Recognise-act Cycle begins, and the instantiation of **start** is added to the Conflict Set (during the instantiation process, the constraints, **(*natnum ?x.0)** and **(*natnum ?y.0)**, were

checked by the application of the principle of Abstract Inter-pattern Consistency). The instantiation of **start** contains the following environment: **(?y.1/?y.0 ?x.1/?x.0)**. The instantiation fires and deposits **(sum-x-y (*plus ?x.1 ?y.1))** in Working Memory.

Working Memory: ((sum-x-y (*plus ?x.1 ?y.1)) (add ?x.0 ?y.0))
Constraint-Set: ((*natnum ?x.0) ∧ (*natnum ?y.0))
Environment: (?y.1/?y.0 ?x.1/?x.0)

On the next cycle, **start** is instantiated, but is rejected because it has already fired. The two positive patterns in **finish** are satisfied by the contents of Working Memory as follows: the first pattern, **(add ?x.2 ?x.2)**, unifies with **(add ?x.0 ?y.0)** and adds the bindings, **(?x.0/?y.0 ?x.2/?x.0)**, to the current environment. The second pattern, **(sum-x-y 0)**, unifies with **(sum-x-y (*plus ?x.1 ?y.1))** generating the equality constraint: **(= (*plus ?y.0 ?y.0) 0)**. The two Abstract Pattern Instantiations are now paired by merging their environments and constraints.

Merged Environment: (?x.0/?y.0 ?x.2/?x.0 ?y.1/?y.0 ?x.1/?x.0)
Merged Constraints: ((= (*plus ?y.0 ?y.0) 0))

It is now time to apply Abstract Negated Pattern Fulfilment. Recall that this process attempts to build a set of constraints (the 'Minimal Negation Nullifier') which would prevent the negated pattern from matching any items in Working Memory. The negated pattern, **~(sum-x-y ?x.2)**, unifies with the WME, **(sum-x-y (*plus ?x.1 ?y.1))**, and produces the Abstract Negated Pattern Instantiation shown below.

Negated Pattern: ~(sum-x-y ?x.2)
WM Element: (sum-x-y (*plus ?x.1 ?y.1))
Bindings: (?x.2/(*plus ?x.0 ?y.0))

The single binding is turned into an equality constraint, **(= ?x.2 (*plus ?x.0 ?y.0))**, and then negated, **¬(= ?x.2 (*plus ?x.0 ?y.0))**; this is the Preliminary Negation Nullifier. The Preliminary Negation Nullifier is then instantiated in terms of the current environment, and passed to the theorem prover, along with the current set of constraints (instantiated). Note that the combined set of constraints is instantiated in terms of the current environment, before passing it to the theorem prover. This explains why the constraint, **(*natnum ?x.0)**, disappears when the current constraint-set is conjoined with the Preliminary Negation Nullifier. When it is instantiated it becomes: **(*natnum ?y.0)**, which then gets deleted because it is a duplicate constraint.

Preliminary Negation Nullifier: ¬(= ?x.2 (*plus ?x.0 ?y.0))
Current Environment: (?x.0/?y.0 ?x.2/?x.0 ?y.1/?y.0 ?x.1/?x.0)
Current Constraint-Set: ((= (*plus ?y.0 ?y.0) 0) ∧ (*natnum ?x.0) ∧ (*natnum ?y.0))

Instantiated Preliminary Negation Nullifier: ¬(= ?y.0 (*plus ?y.0 ?y.0))
Combined Constraint-Set: ¬(= ?y.0 (*plus ?y.0 ?y.0)) ∧ (= (*plus ?y.0 ?y.0) 0)
∧ (*natnum ?y.0))

The theorem prover rejects the combined constraint-set, because it is unsatisfiable. Informally, if **?y.0** is a natural number, which results in zero when added to itself, then **?y.0** can only be zero; but this contradicts the first constraint, which states that **?y.0** is not equal to **?y.0** + **?y.0**. So, the interpreter halts because there are no other rules to fire.

This simple example was carefully chosen to circumvent the need for Conflict Resolution. Before describing how PG handles Conflict Resolution in the abstract domain, we present a brief account of automated theorem proving (section 3.6). The reader familiar with theorem proving can safely skip all of this section apart from 3.6.3 and 3.6.4 which describe PG-specific optimisations.

3.5.7. Summary

Let us now review the work introduced in this section. We opened this section with the observation that any Abstract Interpreter needs to be provided with an Input Specification. Without this constraint, the analysis of even a simple ruleset becomes intractable. This constraint does not limit the applicability of our approach, because it does not require us to provide any more information than that normally built into tutoring systems.

We then described how PG's matching differs from that of an ordinary production system interpreter. In order to handle abstract data objects, PG employs unification, augmented to handle the unification of function expressions with other function expressions or constants. Unification enables us to design abstract counterparts to PGPS's concrete operations.

We noted that PG needs some kind of theorem-proving capability, in order to filter out inconsistent combinations of variable bindings and constraints. Negated patterns are more difficult to handle in the abstract domain. This problem was solved by building an expression which defines the minimal conditions under which the negated pattern is satisfied. This

expression must be checked for consistency with any constraints collected so far. If it violates the current constraint-set, then there is no way that the rule could fire under those constraints.

Firing an Abstract Rule Instantiation is relatively trivial, as most of the work has been carried out during the rule instantiation phase. When a rule fires, the righthand side actions are obeyed; elements deposited in Working Memory typically consisting of uninstantiated variables.

Up until now, we have assumed the presence of a theorem prover, capable of spotting inconsistent constraint-sets. It is now time to detail the features which must be incorporated into such a theorem prover.

3.6. Automated Deduction

This section presents a necessarily limited discussion of a field which has a history spanning some 25 years. We shall concentrate on 'refutation' systems because these are relevant to the task at hand. Such systems prove a theorem by first negating it, and then showing that its negation leads to a contradiction. PG's requires a theorem prover that can spot 'inconsistent' constraint-sets, i.e. ones which are contradictory. This role suggests the use of a theorem prover geared to finding contradictions. Thus, the refutation method seems ideally suited to the job.

Our only claim with regard to our choice of theorem prover is that a refutation procedure is particularly pertinent to PG's needs. Compared to the best theorem provers available today, ours is rather impoverished. Our sole criterion in designing PG's theorem prover was that it be powerful enough to cope with the models to which PG was to be applied.

3.6.1. Resolution Theorem Provers

In the field of Automated Deduction, the term 'resolution' refers to an approach in which the theorem prover, rather than try to prove a theorem directly, determines its validity by demonstrating the *unsatisfiability* of its negation. The rationale for this procedure is simple: if

we can show that the negation of some theorem is false, then it follows that the theorem must be true. To take a completely trivial example, if we know **HUMAN(MARY)** to be true, and wish to prove the theorem **HUMAN(MARY)**, then by negating the theorem, **¬HUMAN(MARY)**, we hit a contradiction - there is no way that *both* of the propositions **HUMAN(MARY)** and **¬HUMAN(MARY)** can be true.

Most resolution theorem provers work on formulae expressed in 'clause form'. Notable exceptions are those of Murray (1982), and Stickel (1982). A 'clause' is a disjunction of 'literals'; for example, **EVEN(2)** ∨ **¬EVEN(2)** is a clause containing two literals, where the first is termed a 'positive literal' and the second a 'negative literal'. A clause can contain zero literals, in which case it is referred to as the 'empty clause', ∅. The empty clause is taken to denote 'false'; intuitively, a disjunction of zero literals cannot be true, so it must be false.

A database in clause form consists of a *set* of clauses. The clauses in this set are implicitly *conjoined*, that is the set of clauses: $\{C_1,...,C_i\}$ is equivalent to: $(C_1 \wedge ... \wedge C_i)$. The resolution procedure works by choosing a pair of clauses, which contain 'complementary' literals, from the current set. The two clauses are resolved with one another to yield a 'resolvent', or 'derived' clause. A pair of literals are 'complementary' if one is positive, the other is negative, and they unify with one another. To derive the resolvent, the resolving clauses are disjoined with one another, and the pair of complementary literals are removed. The substitutions, created during the resolution of the pair of literals, are applied to the other literals in the derived clause. In fact, in full resolution several literals in a clause can be collapsed in one step.

The following example conveys some of the flavour of a resolution refutation proof. This example has been chosen because it is the kind of problem which PG is often required to solve. Consider a situation in which the Input Specification states that the variable, **?x.0**, is even, and PG is trying to instantiate a rule which contains the constraint: **(*odd ?x.1)**, where **?x.1** is bound to **?x.0**. During the process of checking Abstract Inter-pattern Consistency it conjoins the two constraints and passes them to the theorem prover, which then checks their consistency. If the theorem prover derives ∅, then the constraint-set must be incompatible with PG's axiomatisation of the domain. The substitutions in the current environment are applied to the pair of constraints **(*even ?x.0)** and **(*odd ?x.1)** to give **(*even ?x.0)** and **(*odd ?x.0)**. This pair corresponds to the assertion that there *exists* an **?x.0** which is both even and odd, rather than the assertion that *every* **?x.0** is even and odd. This is because the constraints define the conditions which must be *satisfied* for Abstract

Interpretation to continue down the current path. The constraints are satisfiable provided that there is at least one way of validly instantiating the variables therein. For this reason, the variables in the constraint-set are always existentially quantified, thus the above pair of constraints corresponds to the Predicate Calculus formula: $\exists(x)$ **EVEN(x)** $\wedge$ **ODD(x)**.

Because clause form does not include quantifiers, it is important not to loose this information when converting from Predicate Calculus to clause form. During the conversion process, universally quantified variables are left as they are (in other words, any variables in a clause are implicitly universally quantified). However, existentially quantified variables are converted into 'skolem functions' by a process known as 'skolemisation'. Skolemisation preserves the unsatisfiability of the original formula, so for resolution it is a sound transformation to make. Skolemising the above formula, $\exists(x)$ **EVEN(x)** $\wedge$ **ODD(x)**, gives the pair of clauses: **EVEN(sk)** and **ODD(sk)**. The term **sk** is a 'skolem constant', i.e. a skolem function of no arguments.

In fact, if we have our theorem prover adopt the convention that any symbol beginning with a question mark is a skolem constant, then we can keep the constraints pretty much as they are. PG need only apply a minor syntactic transformation to the predicates, by removing the asterisk prefix, and converting the predicate to upper case. Thus, the pair of constraints, **(*even ?x.0)** $\wedge$ **(*odd ?x.0)**, becomes **EVEN(?x.0)** $\wedge$ **ODD(?x.0)**.

We will assume that PG has the following single axiom expressing some of the properties of the predicates **EVEN** and **ODD**:

$\forall(x) \neg(\text{EVEN}(x) \wedge \text{ODD}(x)) \wedge (\text{EVEN}(x) \vee \text{ODD}(x))$.

The above axiom is intended to express the fact, for all **x**, **x** cannot be *both* even and odd, but it is one or the other. Converting to clause form yields the following pair of (implicitly conjoined) clauses:

$\neg\text{EVEN}(x) \vee \neg\text{ODD}(x), \text{EVEN}(x) \vee \text{ODD}(x)$

We are now ready to find out whether the constraint pair, **EVEN(?x.0)** $\wedge$ **ODD(?x.0)**, is inconsistent with this set of clauses. The proof of the unsatisfiability of this set of clauses is shown in figure 3-1.

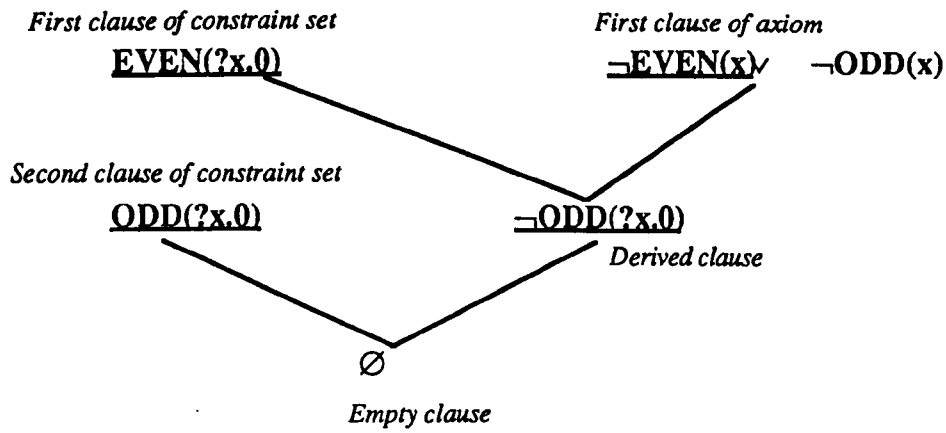


Figure 3-1 — Proof of the unsatisfiability of a constraint-set

The above description of resolution is only the sketchiest of accounts. The interested reader is referred to Nilsson (1980) for a very readable introduction to resolution theorem proving.

PG's theorem prover is a Connection Graph Resolution Theorem Prover (Kowalski, 1975), and employs the Set of Support Strategy. The connection graph is basically an implementation technique which improves performance by indexing potentially resolvable literals. The set of support strategy is a restriction of resolution which requires that at least one of each pair, of the about to be resolved clauses, be a member of the negated theorem or one of its descendants. This restriction is completeness preserving, and has the effect of producing a more goal-directed proof. It is particularly suited to the theorem-proving tasks generated by PG. As we shall see in Chapter 4, the incremental nature of PG's theorem-proving requirements allows it to take advantage of the set of support strategy, and restrict the negated theorem to a very small number of clauses.

3.6.2. Completeness Issues

Resolution is *complete* in the sense that if a set of clauses is unsatisfiable, then the empty clause will eventually be derived. However, it is possible to violate completeness by adopting an incomplete control strategy. For example, if one restricts resolution so that at least one of the pair of resolving clauses comes from the base set, then some proofs will be closed to us (this form of resolution is termed 'Input Resolution'). To preserve completeness, it is

important that the theorem prover be able to resolve two *derived* clauses with one another. For example, the programming language Prolog embodies a resolution theorem prover which employs an *ordered input resolution* control strategy ('ordered' because the clauses are ordered, and 'input' because one of the parent clauses of the resolvent is an input clause, i.e. it is taken from the base set). Although Input Resolution is complete for Horn clauses, Prolog is not, because of its *unbounded* depth-first search strategy (Caferra et al (1984) show how to add completeness to Prolog).

Incompleteness does not always detract from a theorem prover's utility. For one, restrictive control strategies lead to much greater efficiency. Although resolution is complete, for some problems a resolution theorem prover may never find a proof, because its obsession with completeness leads to an unmanageably-large search space. Thus, though one may lose completeness, restricting the search strategy can sometimes provide the theorem prover with the added mileage required to find a solution. In fact, many of the problems tackled in AI are too large to axiomatise fully, and so the program must reason from incomplete knowledge. In such cases, there is little point in choosing a complete theorem prover to tackle an incomplete set of axioms.

3.6.3. Constraint Simplifications

Because we have chosen to use a resolution theorem prover, the constraints carried from cycle to cycle are kept in clause form. The constraint-sets sometimes contain redundant clauses. There are a number of simplification strategies which can be applied to the constraint-set, without affecting its unsatisfiability. PG uses the following three: tautology elimination, elimination of subsumed clauses, and procedural attachment. Clauses which are tautologous (e.g. $P \vee \neg P \vee Q$) are always true, and so cannot contribute to a refutation; thus, they can be eliminated without affecting the falsity of the constraint-set. If a clause, **A**, *subsumes* some other, **B**, then **B** can be removed, because it requires at least as many (and usually more) resolutions to eliminate **B** as it does to eliminate **A**. A clause, **A**, subsumes another, **B**, if there exists a substitution which makes **A**'s literals a subset of those in **B**. For example, the clause: $P \vee \neg Q$ subsumes the clause: $P \vee \neg Q \vee R$.

The final simplification strategy takes advantage of the fact that most constraints can be evaluated, if their arguments are ground. This happens remarkably often during Abstract

Interpretation, as illustrated in the following example. Consider a situation in which the LHS pattern, **(?n)**, is unified with the WME, **((?x ?y))**, giving the binding, **?n/(?x ?y)**. The variable, **?n**, has an associated constraint, **(*natnum ?n)**, which specifies that **?n** must be a natural number. Although **?n**'s binding is not fully ground, it is still sufficiently ground for us to see that it is not a natural number (it is a list of two elements). Therefore, it can be eliminated from the clause. In this particular case, there are no other literals in the clause, so we end up with the empty clause. Thus, this particular binding of **?n** is invalid. In the vast majority of cases where procedural attachment is used, it is performing the function of a 'type checker' (as in the above case, where the binding is rejected because it is of the wrong *type*). If **?n** were bound to **5**, then evaluating the predicate would return 'true'. In this case, the whole clause can be eliminated as it cannot contribute to the unsatisfiability of the constraint-set (the clause is a tautology).

In performing such simplifications, the following question arises. Should the simplified set of constraints be carried over to the next cycle, or should we only apply the simplifications before running the theorem prover, and then reinstate the original unsimplified set of constraints? If it were the case that the constraints are only used to ascertain the validity of solution paths through the production system, then there would be no point in keeping the extra redundant literals and clauses. However, in PG the constraint-set is intended to be used in generating a problem. Therefore, we need to satisfy ourselves that the simplifications will not lose information which could be used to generate a CP.

We argue that tautological and subsumed clauses can be permanently discarded, because they provide *no* CP-generation information. A tautological clause is *always* true, no matter what instantiating members we choose from the domain, therefore, as a constraint it is useless - it tells us nothing about which members to choose from that domain. Clearly, in the following example, the second (tautological) clause is redundant, as it evaluates to true for any possible natural number we might choose.

$$\text{NATNUM}(x) \wedge (\text{PRIME}(x) \vee \neg\text{PRIME}(x) \vee \text{EVEN}(x))$$

Similarly, a subsumed clause can be shown to be redundant because we must first satisfy the *subsuming* clause before the whole clause set can be true. Now, if the subsuming clause is true, it follows that the *subsumed* clause must also be true. Any clause instantiations which are true of the subsumed clause but not the subsuming clause, are of no import, because we are forced to find an instantiation which satisfies the latter. This notion is illustrated by the pair of clauses below, where the first clause subsumes the second.

$$EVEN(x) \wedge (EVEN(x) \vee PRIME(x))$$

The literal, **PRIME(x)**, is superfluous because we must find an **x** which satisfies the predicate **EVEN**. Any such **x** is guaranteed to satisfy the second clause. There is no point in using the literal, **PRIME(x)**, to generate an **x**, because it will ultimately have to satisfy the predicate **EVEN**. Any **x** which satisfies the second clause by virtue of the fact that it is **PRIME** but not **EVEN**, will ultimately be discarded because it cannot satisfy the subsuming clause, **EVEN(x)**.

3.6.4. Improving the Set of Support Strategy by Saving Satisfiable Constraint-sets

Recall that the set of support strategy divides the current set of clauses into those that derive from the negated theorem (termed the 'set of support'), and those that do not. It requires that every resolution involve at least one clause from the set of support, and thereby improves performance by restricting the set of potentially resolvable clause pairs. Now, if we know that some subset of the negated theorem is true, then that subset can be added to the set of axioms. This reduction in the number of clauses in the negated theorem further improves the beneficial effects of the set of support strategy. The incremental nature of PG's theorem-proving tasks, allows it to take advantage of just such a negated-theorem-dividing strategy. As each Abstract Instantiation fires, the set of constraints either gets larger, or stays the same size. So, if the set of constraints is currently, C_i, then on the next cycle, **j**, it will be C_j, where $C_j = C_i \cup C_{new}$ (C_{new} being the new constraints, local to the fired instantiation). On the face of it, C_j should form the negated theorem, to be passed to the theorem prover. However, we already know that C_i is consistent, as it was checked on the previous cycle. Thus, the clauses in C_i can be viewed as axioms. The negated theorem need only consist of the clauses in C_{new}; those in C_i can legitimately be added to the set of axioms.

This augmentation of the algorithm is only worthwhile if one is using a theorem prover which incorporates the set of support refinement (or at least a similar division between axioms and potentially false clause sets). However, in general, there is much potential for optimisations which take advantage of the incremental nature of PG's theorem proving tasks. The incremental nature of the algorithm means that it will usually be worth caching results as the search space unfolds.

After initial runs of PG, it became clear that it is often repeatedly trying to prove the same (or a very similar) theorem. There are many reasons for this. For one, if there is some overlap between the two models (in terms of the constraints generated), then PG will be trying to prove similar theorems for each model. Also, because PG recomputes the Conflict Set from scratch on each cycle, it follows that any instantiation which survives from cycle to cycle, will require exactly the same consistency checking on each cycle. Finally, the 'exclusion clauses' (see next section for a description of these clauses) generated for a set of instantiations are typically very similar to one another.

To reduce redundant effort, PG incorporates two simple clause-set-caching mechanisms; the first pertains to consistent clause-sets. PG caches (i.e. stores for future use) any clause-set which is found to be consistent. Clearly, any subsequent clause-set, which contains exactly the same clauses, is bound to be consistent. However, more generally, any clause-set which is a subset of a cached clause-set must also be consistent.

The second clause-set-caching strategy deals with inconsistent clause-sets. If some clause-set is inconsistent, then it follows that all of its supersets are. In fact, any clause-set which is a superset of the actual clauses (in the base set) which led to a refutation, must also be inconsistent. The clauses which underpin a refutation are easily obtained by following the refutation back to its roots. Note that the 'base set' consists of the negated theorem and the set of consistent clauses collected so far (in our earlier example, the base set was C_j). Thus, PG need only cache those clauses at the roots of the refutation.

3.7. Abstract Conflict Resolution

The abstract versions of the conflict resolution principles are applied in much the same way as their concrete counterparts. Abstract instantiations fired on previous cycles are easily identified. Abstract Recency is also simple to compute, because PG keeps an abstract version of Working Memory. Abstract Specificity is defined in terms of the rules themselves, and is independent of the contents of working memory; thus, its definition is identical to its concrete counterpart. Thus, it would appear that abstract conflict resolution is trivially equivalent to the concrete process. However, although the conflict resolution principles are easily applied to

abstract instantiations, the *overall* process of conflict resolution is subtly different in the abstract domain.

The instantiations in the conflict set represent possible solution paths which the model can follow; conflict resolution chooses one path to follow. This regimen is fine in concrete domains, but in the abstract domain the program must consider instances where the winning rule cannot actually be instantiated. In such cases, the runner-up takes its place. Consider a situation in which there are two abstract instantiations in the conflict set (say I_1 and I_2), and I_1 is the winner after applying Abstract Recency to the conflict set (where I_i is an instantiation of the rule R_i). In the concrete domain, there may be instances where R_2 is satisfied by Working Memory, whilst R_1 is not. Under these conditions, I_2 will fire because there is no I_1. If the Abstract Interpreter ignores this possibility, then it will overlook paths which the model might follow; in other words, *completeness* is sacrificed.

To preserve completeness, PG adopts an analogous solution to that employed for dealing with negated patterns. It builds a set of constraints which define the conditions under which R_1 would fail to be instantiated (we call this set an 'exclusion clause'). The exclusion clause is added to the path emanating from I_2. In effect, the exclusion clause states that this path can only be followed when the exclusion clause is true. If the exclusion clause is incompatible with the constraints already accumulated, then I_2 can never vanquish I_1, and so must be discarded.

3.7.1. Two Simple Conflict Resolution Examples

In the concrete domain, conflict resolution chooses a *unique* instantiation to fire, and only that path is followed subsequently. However, in the abstract domain, the conflict resolution principles can, in general, only be used to *order* the instantiations in the Conflict Set. This ordering process typically enables the program to reject competing instantiations altogether (i.e. when the ordering process yields unsatisfiable exclusion clauses). Nevertheless, the program must cater for cases where the exclusion clauses are satisfiable. In such cases, each satisfiable instantiation forms a choice point in the search for an I/O mapping.

Before presenting the conflict resolution algorithm in detail, we present a couple of simple examples of this procedure, one where the competing instantiation can be rejected altogether,

and one where it cannot. In the first example, the rules **r1** and **r2** are both satisfied by Working Memory, and yield the instantiations I_1 and I_2.

Current Working Memory: ((number ?a.0))
Current Global Constraint-Set: ((*even ?a.0))

```
(define-rule r1
 if (number ?x) & (*even ?x)
 then ...)

(define-rule r2
 if (number ?x)
 then ...)
```

Abstract Rule Instantiation I_1:

New Bindings: (?x.1/?a.0)
New Constraints: ((*even ?x.1))

Abstract Rule Instantiation I_2:

New Bindings: (?x.1/?a.0)
New Constraints: 0

The rule **r1** is more specific than **r2**, therefore the Conflict Set is ordered as follows: $<I_1,I_2>$. Nevertheless, there may be concrete cases where **r2** is instantiable when **r1** is not. This is precisely when the constraints of I_2 are satisfied whilst those of I_1 are not, i.e. when **~(*even ?x.1)**. However, we already know (from the constraints collected so far) that **(*even ?a.0)**. This is incompatible with the exclusion clause **~(*even ?x.1)**, because **?x.1** is bound to **?a.0**. Thus, we can conclude that, given the current set of global constraints, **r2** can *never* fire if **r1** is instantiated.

The second example differs from the first only in the fact that the current global constraint-set is empty, in other words **?a.0** is not constrained to be ***even**.

Current Working Memory: ((number ?a.0))
Current Global Constraint-Set: 0

Abstract Rule Instantiation I_1:

New Bindings: (?x.1/?a.0)
New Constraints: ((*even ?x.1))

Abstract Rule Instantiation I_2:

New Bindings: (?x.1/?a.0)
New Constraints: 0

As in the first example, the exclusion clause for I_2 is **~(*even ?x.1)**, but this time it does not violate any of the constraints collected so far. The two instantiations are equally-valid

candidates for firing, and explicitly define the conditions under which they are eligible for firing, as shown below.

Abstract Rule Instantiation I_1:

Bindings: (?x.1/?a.0)
Constraints: ((*even ?x.1))

Abstract Rule Instantiation I_2:

Bindings: (?x.1/?a.0)
Constraints: ((~ (*even ?x.1)))

In a sense, the ~*even constraint in I_2 is implicit in the ruleset as a whole. In order not to lose such implicit constraints when favouring one path over another, Abstract Interpretation must unravel them and explicitly incorporate them in the divergent paths.

3.7.2. The Abstract Conflict Resolution Algorithm

Conflict resolution begins by applying Refractoriness to the instantiations in the Conflict Set, deleting any which have fired on previous cycles. The remaining instantiations are sifted through the algorithm, described in this section, and any which make it through are eligible for firing. The algorithm takes the current state as its primary argument, and returns a list of valid, abstract successor states. The conflict resolution principles are applied repeatedly to the instantiations until all conflicts have been resolved. On each pass through conflict resolution, two sets of instantiations are derived. The first consists of the winning instantiations (i.e. those which make it through conflict resolution and cannot be resolved further); the second consists of those instantiations which were rejected by conflict resolution. Each winning instantiation (W_i) is paired off against the other winning instantiations and any other winning instantiations collected on previous passes (termed 'Previous Victors'). The constraint-sets in the other winning instantiations and Previous Victors are then individually negated. Each negated set is then conjoined to form the exclusion clause for the winning instantiation (W_i). This exclusion-generation procedure is carried out for each of the winning instantiations. The winning instantiations, complete with exclusion clauses, are then added to the list of Previous Victors. The program is now ready to commence another pass through the Conflict Set, which now consists only of the losing instantiations from the previous pass. The process terminates once all of the instantiations have been processed and passed on to the list of Previous Victors. Each instantiation, so processed, can then be converted into a new state and added to the

search tree. One can save some computational effort by using the exclusion clause of the *most recently computed* Previous Victor, in place of the negation of the constraints of all of the Previous Victors. The two methods are equivalent. We have presented the less efficient of the two, because in our opinion the semantics are clearer.

The above verbal description is made a little more precise by the following pseudo-Prolog definition.

```
/* The CurrentState is a structure which holds the conflict set, current bindings, and accumulated
constraints. 'SuccessorStates' is an output argument. */
generateSuccessorStates(CurrentState,SuccessorStates) <-
  stateConflictSet(CurrentState,ConflictSet) &  /* Accesses the Conflict Set of the current state. */
  filterConflictSet(CurrentState,ConflictSet,[],[],SuccessorStates).

/* Only the final argument is an output variable.
   Conflict resolution partitions the current Conflict Set into a set of winners and a set of losers.
   The winners are turned into new states by adding an exclusion clause to each one.
   The losers now form the new Conflict Set, ready for the next pass. The whole process
   ends when the Conflict Set is empty. */
filterConflictSet(_,[],_,EligibleStates,EligibleStates).
filterConflictSet(CurrentState,ConflictSet,PreviousVictors,StatesSoFar,EligibleStates) <-
  /* The predicate, applyConflictResolution, applies the conflict resolution principles to the Conflict
     Set, and returns a list of the winning instantiations and a list of the losing ones. */
  applyConflictResolution(ConflictSet,WinningInstantiations,LosingInstantiations) &
  createNewStates(WinningInstantiations,PreviousVictors,CurrentState,NewStates) &
  append(NewStates,StatesSoFar,StateCollection) &
  append(WinningInstantiations,PreviousVictors,NewVictorsList) &
  filterConflictSet(CurrentState,LosingInstantiations,NewVictorsList,StateCollection,EligibleStates).

/* The final argument is an output variable.
   Each instantiation has an exclusion clause attached to it, and is converted into a new state
   by copying the contents of the current state, and adding in the bindings, constraints and
   exclusion clause of the modified instantiation. The modified instantiation is also stored in
   the new state, ready for firing on the next cycle. */
createNewStates(Instantiations,PreviousVictors,CurrentState,NewStates) <-
  addExclusionClauses(Instantiations,Instantiations,PreviousVictors,CurrentState) &
  turnInstantiationsIntoNewStates(Instantiations,CurrentState,NewStates).

/* For each instantiation add in an exclusion clause based on the constraints in the other
   instantiations and previous victors. */
addExclusionClauses([],_,_,_).
addExclusionClauses([I|RestOfInsts],Instantiations,PreviousVictors,CurrentState) <-
  remove(I,Instantiations,CompetingInstantiations) &
  addInExclusionClause(I,CompetingInstantiations,PreviousVictors,CurrentState) &
  addExclusionClauses(RestOfInsts,Instantiations,PreviousVictors,CurrentState).

/* This clause is passed an instantiation, 'I', a list of the competing instantiations, a list of the previously
victorious instantiations, and the current state. It modifies 'I' so that it's exclusion clause explicitly excludes
the competing instantiations and previous victors. */
addInExclusionClause(I,CompetingInstantiations,PreviousVictors,CurrentState) <-
  append(CompetingInstantiations,PreviousVictors,AllCompetitors) &
  collectNegatedCompetitorConstraintSets(AllCompetitors,CurrentState,NegatedConstraints) &
  setExclusionClause(I,NegatedConstraints).
```

The above algorithm does not specify how to convert the constraints of competing instantiations into an exclusion clause. On the face of it, all one need do is collect the

constraints in each instantiation, negate each set, and conjoin them. For example, consider a situation in which an instantiation, **I**, has the following two competitors (C_1 and C_2):

Constraints of C_1: EVEN(?a) ∧ EVEN(?b)

Constraints of C_2: EVEN(?a) ∧ ODD(?c)

If we negate each set and conjoin them, we obtain the following exclusion clause:

¬(EVEN(?a) ∧ EVEN(?b)) ∧ ¬(EVEN(?a) ∧ ODD(?c))

which simplifies to the following clause-form expression:

(¬EVEN(?a) ∨ ¬EVEN(?b)) ∧ (¬EVEN(?a) ∨ ¬ODD(?c))

The instantiation, **I**, has the constraint-set: **(EVEN(?a))** already, because it is a global constraint, inherited by all of the instantiations in the Conflict Set (this explains its presence in both C_1 and C_2). Thus, the combined constraint-set of **I** will be as follows, when the exclusion clause is added:

EVEN(?a) ∧ (¬EVEN(?a) ∨ ¬EVEN(?b)) ∧ (¬EVEN(?a) ∨ ¬ODD(?c))

This is perfectly valid, but somewhat clumsy. Clearly, the presence of the disjunct, **¬EVEN(?a)**, in clauses 2 and 3, is redundant. Given the first clause, **EVEN(?a)**, it is clear that **¬EVEN(?a)** can never hold, and so may as well be discarded. The following exclusion clause is equivalent to, and subsumes its unwieldy cousin:

EVEN(?a) ∧ ¬EVEN(?b) ∧ ¬ODD(?c)

It is obtained by first removing the global constraints, *before* we negate those in the competitor. If we do this to the constraint-sets of C_1 and C_2 we obtain:

Constraints of C_1: EVEN(?b)

Constraints of C_2: ODD(?c)

which yields the exclusion clause:

¬EVEN(?b) ∧ ¬ODD(?c)

The exclusion clause must also include constraints which are *implicit* in the competitors. These are the equality constraints implicitly defined by the process of pattern matching. Consider the following two instantiations, I_1 and I_2, created by matching **r1** and **r2** with the same WME.

Current Working Memory: ((?a.0 likes ?b.0))

```
(define-rule r1
 if (?x likes ?x)
 then ...)

(define-rule r2
 if (?x likes ?y)
 then ...)
```

Abstract Rule Instantiation I_1:

New Bindings: (?a.0/?b.0 ?x.1/?a.0)

Abstract Rule Instantiation I_2:

New Bindings: (?y.1/?b.0 ?x.1/?a.0)

The rule, **r1**, embodies an implicit equality constraint between the first and third sub-elements of its LHS condition. Were we to build an exclusion clause for **r2** based solely upon the explicit constraints in **r1**, then we would lose the crucial information that **r2** (which is less specific than **r1**) can only take precedence if the first and third sub-elements, of the WME matched to, are not equal. Therefore, the bindings of the competitors must be converted into equality constraints, and added to the competitor's constraint-set to form the exclusion clause. Performing this modification to I_1 yields the following steps on the way to building I_2's exclusion clause.

Equality Constraints of I_1: $(= ?a.0\ ?b.0) \wedge (= ?x.1\ ?a.0)$

Negated I_1 Constraints: $\neg(= ?a.0\ ?b.0) \vee \neg(= ?x.1\ ?a.0)$

On the face of it, the exclusion clause contains a redundant disjunct, $\neg(= ?x.1\ ?a.0)$; but this is illusory. In fact, it should not be there at all - it is invalid. The variables named **?x** in **r1** and **r2**, should not be confused with one another. The fact that they have the same name is mere coincidence. To avoid confusion they should be standardised apart (say by prefixing the LHS variables with the rule name), as shown below.

Equality Constraints of I_1: $(= ?a.0\ ?b.0) \wedge (= ?r1x.1\ ?a.0)$

Negated I_1 Constraints: $\neg(= ?a.0\ ?b.0) \vee \neg(= ?r1x.1\ ?a.0)$

Now there is no longer a naming confusion between the two **?x.1**'s. If we add the exclusion clause to I_2, then we will obtain the following version of I_2.

Abstract Rule Instantiation I_2:

New Bindings: (?r2y.1/?b.0 ?r2x.1/?a.0)

Exclusion Clause: ¬(= ?a.0 ?b.0) ∨ ¬(= ?r1x.1 ?a.0)

Unfortunately, this situation is just as problematic. The variable, **?r1x.1**, is meaningless in the context of I_2, as it only gets bound when following the I_1 path. Fortunately, one simple observation allows us to circumvent all of these problems. Let us recall why we were creating these equality constraints in the first place. Our purpose was to make explicit any implicit constraints on the elements in Working Memory (for example, the two **?x**'s in **r1** constrain **?a.0** and **?b.0** to be equal to one another). However, not all of the bindings of **r1** serve this function - we can take for granted the fact that, in the abstract, **?x.1** will always be able to match **?a.0**. Such bindings are always satisfiable, and so cannot usefully form part of an exclusion clause. It is easy to spot such bindings amongst the equality constraints - they always contain one of the LHS variables of the competing instantiation. Thus, of the two equality constraints from I_1, **(= ?a.0 ?b.0)** and **(= ?x.1 ?a.0)**, only the first is a true inter-variable equality constraint; therefore, the second constraint should be left out of the exclusion-clause-building process:

Inter-variable Equality Constraints of I_1: (= ?a.0 ?b.0)

Negated I_1 Constraints: ¬(= ?a.0 ?b.0)

Abstract Rule Instantiation I_2:

New Bindings: (?y.1/?b.0 ?x.1/?a.0)

Exclusion Clause: ¬(= ?a.0 ?b.0)

We can now finish setting out the exclusion-clause-building algorithm in pseudo-Prolog.

```
/* Collects an exclusion set from each competitor and returns their conjunction. */
collectNegatedCompetitorConstraintSets([],_,ExclusionClause).
collectNegatedCompetitorConstraintSets([CompetitorIRest],CurrentState,ExclusionClause) <-
  generateExclusionSet(Competitor,CurrentState,ExclusionSet) &
  collectNegatedCompetitorConstraintSets(Rest,CurrentState,ExclusionSetsOfRest) &
  conjoin(ExclusionSet,ExclusionSetsOfRest,ExclusionClause).

/* Computes a set of exclusion constraints.
   The constraints of the competitor are collected together with any equality constraints
   implicit in the competitor's bindings. Only those constraints which are not global
   (i.e. common to all instantiations in the current state) are negated and returned. */
generateExclusionSet(Competitor,CurrentState,ExclusionSet) <-
  collectLHSvars(Competitor,LHSvariables) & /* Accesses LHS variables of instantiation. */
  generateCompetitiveConstraints(Competitor,LHSvariables,CompetitiveConstraints) &
  stateConstraints(CurrentState,GlobalConstraints) & /* Accesses the constraints collected so far. */
  negateLocalConstraintsOnly(CompetitiveConstraints,GlobalConstraints,ExclusionSet).

/* The bindings are turned into equality constraints. Those containing one of the LHSvars
   are discarded. The equality constraints not so discarded are conjoined with instantiation's
   constraints, and returned. */
generateCompetitiveConstraints(Instantiation,LHSvars,ReturnedConstraints) <-
  instantiationEnv(Instantiation,Bindings) & /* Accesses the bindings of an instantiation. */
  turnBindingsIntoEqualityConstraints(Bindings,EqualityConstraints) &
  removeThoseContainingAnLHSvar(EqualityConstraints,LHSvars,FilteredEqualityConstraints) &
  instantiationConstraints(Instantiation,Constraints) & /* Accesses constraints of an instantiation. */
  conjoin(FilteredEqualityConstraints,Constraints,ReturnedConstraints).

/* Only negates those constraints which are not already part of the current state. */
negateLocalConstraintsOnly(CompetitiveConstraints,GlobalConstraints,NegatedConstraints) <-
  intersection(CompetitiveConstraints,GlobalConstraints,CommonConstraints) &
  remove(CommonConstraints,CompetitiveConstraints,LocalConstraints) &
  negateConstraintSet(LocalConstraints,NegatedConstraints).
```

Note that our algorithm for applying conflict resolution during Abstract Interpretation is completely general and does not rely on the particular strategy chosen for PGPS. This is because it accurately models the semantics of the conflict resolution process as a procedure in which a set of conflict resolution rules is applied to progressively filter the conflict set to a singleton. In the interests of parsimony, we decreed that it is an error for a ruleset to enter a state in which the conflict resolution strategy terminates with more than one instantiation in the conflict set. This in no way limits the applicability of our approach; one can cater for this case by having PG expand each remaining instantiation in turn.

Our description of the Abstract Interpretation of production systems is now complete. In the following section, we describe some minor augmentations which enable us to Abstractly Interpret productions systems in *pairs* rather than one at a time.

3.8. Paired Abstract Interpretation

As described so far, Abstract Interpretation is carried out on a single model. In fact, PG interprets the pair of candidate models in tandem. Recall that its overall goal is to find a *single* input for which each model produces a *different* output. A naive way to achieve this goal is to find *all* of the I/O mappings for each model, and to then sift through these until a pair of mappings, with an overlapping input but non-overlapping output, is found. However, it is more economical to only associate the paths in one model with those paths in the other which have overlapping abstract inputs. This strategy enables PG to discard unpromising paths early on. For example, if the current path in some model specifies that the input is even, whilst another path in the competing model specifies that it is odd, then there is no point developing these paths further. They will never yield a Critical Problem, because there is no concrete input which is both even and odd.

As PG expands each state, it keeps a record of the pairs of states (one from each model) which are consistent with one another. A pair of states is said to be consistent if the range of possible instantiations of the input variables overlaps. The range of possible instantiations is defined by the set of constraints collected so far. If the two concrete domains, covered by the constraint-sets, do not overlap, then there is no way of instantiating both abstract inputs with the same concrete input.

The following section outlines the overall control structure of PG. We choose to view PG as performing a search through a space of Abstract Interpretation 'State Pairs', where each State Pair consists of a consistent state from each model. In the following account, we assume the presence of a 'State Chooser', which selects the next state to expand, from those available. In an actual implementation, the State Chooser could select states in such a way as to produce any search strategy one can conceive of. The current implementation of PG adopts a depth-first search strategy, but we claim that a heuristic strategy would lead to much improved performance (see section 4.8.3).

3.8.1. PG's Control Structure

As usual, we present the algorithm in pseudo-Prolog. PG is invoked with a request to find a CP (**findCP**) using a pair of models, and a list of Input Specifications. PG selects the first Input Specification, renames the variables therein, and then tries to find a discriminating output (**findDiscriminatingOutput**) for that Input Specification. If it fails to find a CP for that Input Specification, then it moves on to the next one. The whole process only results in failure if there are no more Input Specifications to try.

```
findCP(Model1,Model2,[InputSpec|RestofInputSpecs],CP) <-
  renameVariables(InputSpec,0,RenamedInputSpec) &  /* Give variables a suffix of zero. */
  findDiscriminatingOutput(Model1,Model2,InputSpec,CP).
findCP(Model1,Model2,[InputSpec|RestofInputSpecs],CP) <-
  renameVariables(InputSpec,0,RenamedInputSpec) &
  ~findDiscriminatingOutput(Model1,Model2,InputSpec,_) &
  findCP(Model1,Model,RestOfInputSpecs,CP).
```

To find a CP for a particular Input Specification, PG parcels out the constraints and initial WMEs therein (**collectWMEsAndConstraints**). These are passed to **initialiseOpenList** which creates an initial state for each of the two models (**initialiseState**), computes the successors of each initial state (**generateSuccessorStates**), and finally returns a list of consistent State Pairs (**pairConsistentStates**). The initial Open list is then used as a basis for the search for a CP (**searchForDifference**).

```
findDiscriminatingOutput(Model1,Model2,InputSpec,CP) <-
  collectWMEsAndConstraints(InputSpec,WMEs,Constraints) &      /* Separates components of
                                                                  InputSpec. */
  initialiseOpenList(Model1,Model2,WMEs,Constraints,InitialOpen) &
  searchForDifference(Model1,Model1,Model2,InitialOpen,CP).

initialiseOpenList(Model1,Model2,WMEs,Constraints,InitialOpen) <-
  initialiseState(Model1,WMEs,Constraints,State1) &
  initialiseState(Model2,WMEs,Constraints,State2) &
  generateSuccessorStates(State1,Successors1) &
  generateSuccessorStates(State2,Successors2) &
  pairConsistentStates(Successors1,Successors2,InitialOpen).

/* The initial state for a model is formed by creating a structure which holds the model,
   the initial WMEs, constraints, and a level marker set to zero.
   These initial data structures are then used to compute the initial Conflict Set. */
initialiseState(Model,WMEs,Constraints,State) <-
  makeState(Model,WMEs,Constraints,0,State) &
  computeConflictSet(State,ConflictSet) &  /* Computes conflict set, as described in section 3.5.4. */
  setConflictSet(State,ConflictSet). /* Sets the Conflict Set slot of a state. */

/* Each state in the first argument position is paired with every state in Successors2 with
   which it is consistent. Consistency is checked by passing the sets of constraints in each
   to a theorem prover capable of finding a contradiction if one exists. */
pairConsistentStates([],_,[]).
pairConsistentStates([S1|Rest1],Successors2,StatePairs) <-
  pairState1WithAllSuccessorsOf2(S1,Successors2,S1Pairs) &
  pairConsistentStates(Rest1,Successors2,StatePairsOfRest) &
  append(S1Pairs,StatePairsOfRest,StatePairs).

pairState1WithAllSuccessorsOf2(_,[],ListofStatePairs).
pairState1WithAllSuccessorsOf2(State1,[State2|Rest2],[StatePair|StatePairsOfRest]) <-
  consistentStatePair(State1,State2) &
  makeStatePair(State1,State2,StatePair) &
  pairState1WithAllSuccessorsOf2(State1,Rest2,StatePairsOfRest).
```

At any given time, PG is concentrating on one of the pair of models (**CurrentModel**). On each cycle of the search, it chooses a State Pair (**chooseStatePair**) which contains the current model in an *unhalted* state. This is because there is no point in trying to expand a state in which the model has halted, as, by definition, the model has ceased execution. If it cannot find a State Pair which contains the current model in an unhalted state, then it switches to the other model (**switchRulesets**) and selects a State Pair in which the other model has not halted. If both models are halted in all of the open State Pairs, then no further processing is possible, and so it returns with failure. Assuming that there is a State Pair containing an unhalted model, then it is selected for expansion (**expandStatePair**).

```
searchForDifference(CurrentModel,Model1,Model2,Open,CP) <-
  chooseStatePair(CurrentModel,Open,StatePair) &
  expandStatePair(StatePair,CurrentModel,Model1,Model2,Open,CP).
searchForDifference(CurrentModel,Model1,Model2,Open,[]) <-
  ~chooseStatePair(CurrentModel,Open,StatePair) &
  switchRulesets(Model1,Model2,CurrentModel,NewCurrentModel) &
  chooseStatePair(NewCurrentModel,Open,StatePair) &
  expandStatePair(StatePair,NewCurrentModel,Model1,Model2,Open,CP).
```

Next, the sole rule in the Conflict Set for the chosen state is fired, and the successor State Pairs are added to the Open list (**fireRuleAndExpandState, fireRule, processResultOfStateExpansion**). If the outputs, produced by firing the rule, allow PG to generate a CP (**canGenerateCP**), then that CP is returned.

```
expandStatePair(StatePair,CurrentModel,Model1,Model2,Open,CP) <-
  fireRuleAndExpandState(StatePair,CurrentModel,Model1,Model2,
                         NewStatePair,NextModel,NewOpen) &
  ~canGenerateCP(NewStatePair,_) &
  searchForDifference(NextModel,Model1,Model2,NewOpen,CP).
expandStatePair(StatePair,CurrentModel,Model1,Model2,Open,CP) <-
  fireRuleAndExpandState(StatePair,CurrentModel,Model1,Model2,
                         NewStatePair,NextModel,NewOpen) &
  canGenerateCP(NewStatePair,CP).

fireRuleAndExpandState(StatePair,CurrentModel,NewStatePair,NextModel,NewOpen) <-
  fireRule(StatePair,CurrentModel,NewStatePair) & /* As described in section 3.5.5. */
  processResultOfStateExpansion(NewStatePair,CurrentModel,Model1,Model2,Open,
                                NextModel,NewOpen).
```

With the rule having fired, and Working Memory, the output list, and the halt status, modified accordingly, PG is now ready to compute the set of successor states. If the current ruleset has halted, then PG switches control to the other model (**currentRulesetHasHalted, switchRulesets**). If the current ruleset has not halted, then compute the successors (**expandStatePair**), and process any outputs generated by the fired rule (**dealWithNewOutputs**).

```
processResultOfStateExpansion(NewStatePair,CurrentModel,Model1,Model2,Open,
                              NextModel,Open) <-
  currentRulesetHasHalted(CurrentModel,NewStatePair) &
  switchRulesets(Model1,Model2,CurrentModel,NextModel).
processResultOfStateExpansion(NewStatePair,CurrentModel,Model1,Model2,Open,
                              NextModel,NewOpen) <-
  ~currentRulesetHasHalted(CurrentModel,NewStatePair) &
  generatedNewOutputs(NewStatePair) &
  dealWithNewOutputs(NewStatePair,Model1,Model2,CurrentModel,Open,NextModel,NewOpen).
processResultOfStateExpansion(NewStatePair,CurrentModel,Model1,Model2,Open,
                              CurrentModel,NewOpen) <-
  ~currentRulesetHasHalted(CurrentModel,NewStatePair) &
  ~generatedNewOutputs(NewStatePair) &
  expandStatePair(NewStatePair,CurrentModel,Open,NewOpen).
```

New outputs are handled by switching to the ruleset with the fewest outputs. This practice is based on the observation that PG need only find the first non-equivalent I/O pair. It is not necessary for PG to compute both output strings, in their entirety, and then compare them to see if there is a difference. It is sufficient for it to compute the outputs for each model, one at a time, stopping either when it finds a non-eqivalent output or when one output string is complete and the other string is longer. Thus, it should be developing the model which, during the search so far, has produced the fewest outputs - there is no point in processing the other model until both models have produced an equal number of outputs.

```
dealWithNewOutputs(NewStatePair,Model1,Model2,CurrentModel,Open,NextModel,NewOpen) <-
  expandStatePair(NewStatePair,CurrentModel,Open,NewOpen) &
  switchToRulesetWithFewestOutputs(Model1,Model2,NewStatePair,NextModel).
```

To expand a state, PG takes the new State Pair (with Working Memory changes recorded therein) and computes the new Conflict Set for the state currently being worked on (**chosenState, computeConflictSet**). The successors of the chosen state are computed, as described in section 3.7.2 (**generateSuccessorStates**). Each State Pair on Open, containing the state just expanded, is collected (**collectParentPairs**) so that each can be removed from the Open list and replaced by a set of new State Pairs, consisting of a successor state and the old companion state.For example, consider a situation where PG has just expanded the state, **s1b**, and obtained the successors, **s1c**, **s1d**, and **s1e**. The Open list currently contains the State Pairs: <s1b,s2a>, <s1b,s2c>, and <s1a,s2a>. Each State Pair containing **s1b** is replaced by a set of State Pairs, where one state is a successor of **s1b**, and the other is a former companion of **s1b**. Thus, the Open list becomes: <s1c,s2a>, <s1c,s2c>, <s1d,s2a>, <s1d,s2c>, <s1e,s2a>, <s1e,s2c>, and <s1a,s2a>. The clauses named **collectParentPairs** collect those State Pairs containing the state just expanded. The job of replacing these with the set of successors, paired with former companions, is carried

out by **replaceEachParentPairWithSuccessorPairs**. Note that it will only pair a given successor with a companion state, if their sets of constraints are consistent with one another.

```
expandStatePair(NewStatePair,CurrentModel,Open,NewOpen) <-
 chosenState(NewStatePair,CurrentModel,StateToExpand) &
 computeConflictSet(StateToExpand,ConflictSet) &
 setConflictSet(StateToExpand,ConflictSet) &
 generateSuccessorStates(StateToExpand,Successors) &
 collectParentPairs(StateToExpand,Open,ParentPairs) &
 replaceEachParentPairWithSuccessorPairs(StateToExpand,ParentPairs,Successors,Open,NewOpen).

collectParentPairs(StateToExpand,[],[]).
collectParentPairs(StateToExpand,[StatePair|RestOfOpen],[StatePair|ParentPairsOfTail]) <-
 containsState(StateToExpand,StatePair) &
 collectParentPairs(StateToExpand,RestOfOpen,ParentPairsOfTail).
collectParentPairs(StateToExpand,[StatePair|RestOfOpen],ParentPairsOfTail) <-
 ~containsState(StateToExpand,StatePair) &
 collectParentPairs(StateToExpand,RestOfOpen,ParentPairsOfTail).

/* The predicate, collectCompanionStates, collects the states which were formerly paired
   with the state just expanded The predicate, listDifference, merely computes the set of items
   which are in its first argument but not its second. */
replaceEachParentPairWithSuccessorPairs(StateToExpand,ParentPairs,Successors,Open,NewOpen) <-
 collectCompanionStates(StateToExpand,ParentPairs,CompanionStates) &
 pairConsistentStates(Successors,CompanionStates,NewPairs) &
 listDifference(Open,ParentPairs,RestOfOpen) &
 append(NewPairs,RestOfOpen,NewOpen).
```

The above algorithm is adequate if one does not want PG to reason about cases where one model is unable to continue, because no rules match Working Memory. Under such conditions, it may be possible to discriminate between the two models, by virtue of the fact that only one model produces an output, because the other model is unable to continue to a point where it would have produced an output. It is not difficult to modify the above algorithm so that such cases are covered. PG does not fully implement this functionality; therefore its derived description of I/O mappings is sometimes not exhaustive. The following two sub-sections outline the required modifications to the above algorithm, and also describes the less general solution employed in PG. We also discuss the pros and cons of incorporating an exhaustive solution to this problem.

3.8.2. Characterising Inconsistent State Pairs

When we say that a pair of states are inconsistent, we are saying that no member of the domain of discourse can satisfy all of the constraints in both states. For example, if one state of the pair includes the constraint EVEN(?x), whilst the other state includes ODD(?x), then the pairing is inconsistent, because no number satisfies both predicates. Access to the required

axioms, together with an adequate inferencing capability, enables PG to spot the contradiction. Couched in the language of sets, a pair of states is inconsistent if the range of instances covered by each, does not intersect. The Venn Diagram in figure 3-2 illustrates this.

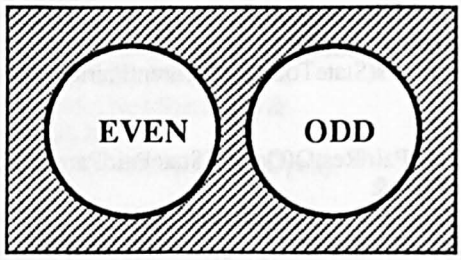


Figure 3-2 — Venn Diagram representation of the sets of even and odd numbers

Thus, if we have two models, where one requires that **?x** be even and the other requires that it be odd, then there is no value of **?x** which would allow both of these models to run. Nevertheless, it is still possible to discriminate between two such models, by choosing either an even or an odd number for **?x**. In either case, one model would produce an output, whilst the other would fail to produce anything at all.

The situation is a little more complex when the two constraint-sets intersect. To illustrate this, we will again make use of Venn Diagrams. We shall assume that three propositions are used by the two models (α, β, and γ), and that they all pertain to the same input variable. We join the process at the point where the Open list contains the single state pair: <s1b,s2a>, **s1b** having the associated proposition 'α'. The state, **s2a**, is chosen for expansion, and, after applying Abstract Conflict Resolution, produces two successors, **s2b**, and **s2c**. The state, **s2b**, has an associated constraint (proposition), β, and **s2c** has the proposition γ, and because it lost to **s2b** during conflict resolution, has the exclusion clause $\neg\beta$. The Venn Diagram for α, β, and γ is shown in figure 3-3.

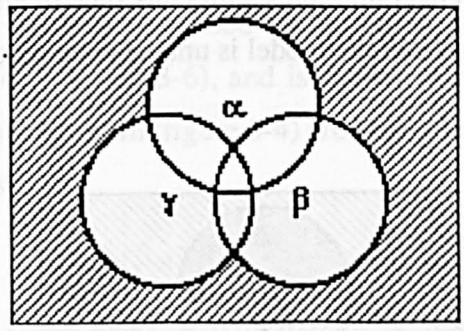
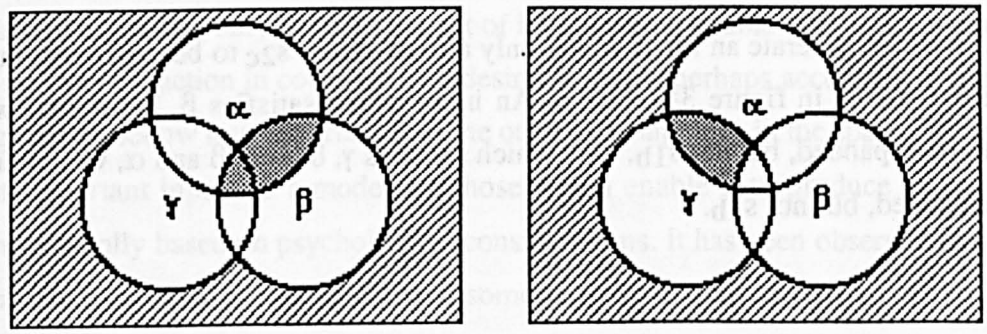


Figure 3-3 — The relationship between the sets denoted by α, β, and γ

If we pair **s1b** with the two successors of **s2a** (as proscribed in **expandStatePair**, in section 3.8.1), then the Open list would be as follows: <s1b,s2b>, <s1b,s2c>. The sets of elements which are consistent with the constraints of each pair, are shown below in figure 3-4. Consistency is represented as a darkly shaded area.



<s1b,s2b>

Constraints: $\alpha \wedge \beta$

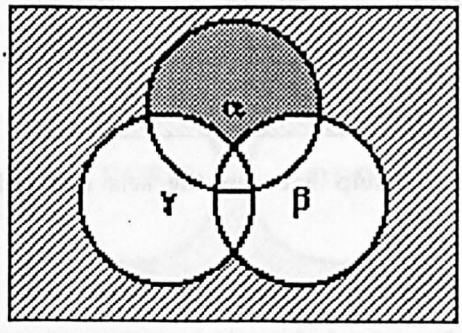
<s1b,s2c>

Constraints: $\alpha \wedge \gamma \wedge \neg\beta$

Figure 3-4 — The elements satisfying the constraints in each State Pair

The algorithm, described in section 3.8.1, would produce the above two State Pairs. However, there are other inputs, which do not lie in these intersections, but could be given to the models. For example, the input represented by: $\alpha \wedge \neg\beta \wedge \neg\gamma$, could be given to both models. Such an input would force the model, represented by **s2b** and **s2c**, to halt, because the constraints are violated. Nevertheless, the model represented by **s1b**, would carry on

processing the input (the Venn Diagram representing this state of affairs, is shown in figure 3-5, where 'NIL' represents the fact that the model is unable to continue).

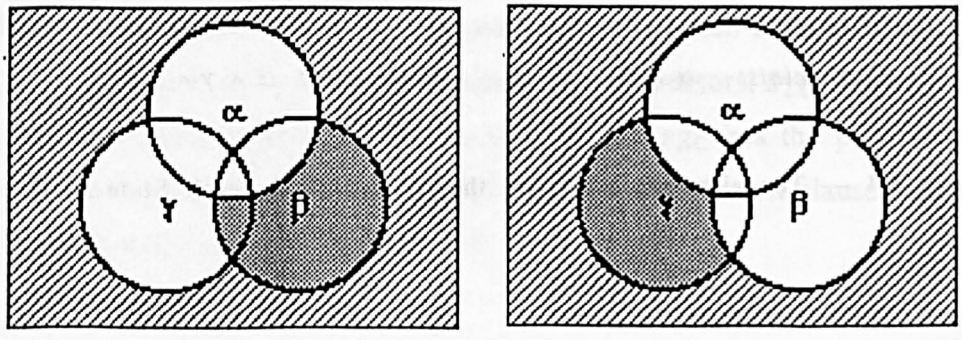


<s1b,NIL>

Constraints: $\alpha \wedge \neg\beta \wedge \neg\gamma$

Figure 3-5 — A set of inputs which only satisfies the state s_{1b}

Similarly, one can generate an input which only allows **s_{2b}** or **s_{2c}** to be expanded, but not **s_{1b}**. These are shown in figure 3-6, below. An input which satisfies β, but not α, would allow **s_{2b}** to be expanded, but not **s_{1b}**. One which satisfies γ, but not β and α, would enable **s_{2c}** to be expanded, but not **s_{1b}**.



<NIL,s2b> *<NIL,s2c>*

Constraints: $\beta \wedge \neg\alpha$ *Constraints:* $\gamma \wedge \neg\alpha \wedge \neg\beta$

Figure 3-6 — Inputs which do not satisfy the state s_{1b}

The abstract space of all inputs is described by the union of all of the Venn Diagrams presented above (figures 3-4, 3-5, and 3-6), and is shown in figure 3-7. Clearly, the state pairs generated by the current algorithm (figure 3-4) are losing information about the space of inputs handled by the models.

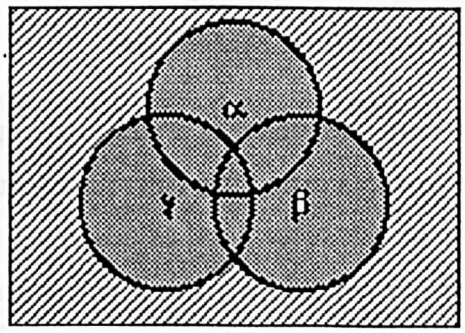


Figure 3-7 — The union of all of the shaded areas from figures 3-4, 3-5, and 3-6

The current algorithm only encodes the set of inputs which enable *both* models to continue processing. Any reduction in coverage is undesirable, but is perhaps acceptable if the lost I/O mappings are somehow less important than the other I/O mappings in the space. We argue that the most important inputs to a model are those which enable it to produce an output. Our argument is wholly based on psychological considerations. It has been observed that, when a student is stuck on some problem, s/he will sometimes try to find some way of extracting an answer, even if his/her algorithm is unable to continue without modification. Repair Theory (Brown & VanLehn, 1980) is based on this notion, where being stuck is termed an 'impasse', and the method of circumventing the impasse a 'repair'. Thus, if we set the student a problem which, according to our current model, will cause them to halt in mid-solution, then we run the risk that they will try to augment the algorithm so that some answer can be found. However, if we set problems which we expect the student to solve, even if erroneously, then there is less of a risk that s/he will modify his/her procedure. A CP (Critical Problem) which causes the student model to output an answer, is more likely to lead to the student remaining faithful to his/her algorithm. Thus, such a CP is preferable to one which leads to a mid-solution halt, because it introduces less noise into the data.

To preserve exhaustiveness, we could use such a preference criterion to partially order the State Pairs on the Open list. The State Pairs, where one model is unable to continue, would

only be be expanded if there were no State Pairs left of the other kind. Our decision to discard the former type of State Pair is an implementation detail, added to reduce search. We shall now describe the exhaustive solution to this problem, together with the more restricted solution employed in PG.

3.8.3. Dealing With Non-Intersecting State Pairs

In order to cover cases where only one model of a pair is able to proceed, the algorithm needs to be augmented so that it produces extra State Pairs representing this situation. Given a State Pair, <s_{1b},s_{2a}>, we compute the successors, as described in section 3.8.1, but also compute successors which encode the conditions under which either s_{1b}, or s_{2a} are forced to halt, because none of their rules are eligible for firing. A state is forced to halt if its constraints are violated. To explicitly violate a state's constraints, we need only negate them. For example, returning to the $\alpha/\beta/\gamma$ example presented earlier, we can represent the situation in which s_{1b} is forced to halt, but s_{2b} and s_{2c} are able to continue, by negating the constraints in s_{1b}, and pairing this state with s_{2b} and s_{2c}. This gives us the two State Pairs shown in figure 3-6. Similarly, one represents the situation in which s_{1b} can be expanded, but s_{2b} and s_{2c} cannot, by negating the constraints in *both* s_{2b} and s_{2c} (we must negate both, because this State Pair represents a situation in which neither s_{2b} nor s_{2c} can be expanded, i.e. both of their constraint-sets are false). This gives us the State Pair shown in figure 3-5. Note that the state 'NIL' is used illustratively to represent the fact that the state cannot be expanded further in the context of its companion; but it still contains all of the information collected so far (information such as the negated constraint-set). It is important to retain this information for CP generation. The problem generator needs to know that these two states are only paired when the constraints of the state, denoted here by 'NIL', are violated.

PG's compromise solution is to only create pairings, with each state forced to halt, when no consistent pairing exists. The even/odd diagram, in figure 3-3, is an example of such a case. A pair of states are inconsistent with one another, if their sets of constraints are non-intersecting. If PG did not adopt the compromise solution, then it would fail to handle such cases.

Thus, in the even/odd example, PG would pair each state with its companion in a forced-halt state. However, in the $\alpha/\beta/\gamma$ example, it would not create such pairings, as there exist consistent pairings (figure 3-4).

We will now describe how PG converts a pair of I/O mappings into an abstract description of the set of CPs.

3.9. Generating the Abstract CP Description

Thus far, we have described how to take a pair of production rule models and derive an abstract description of the I/O mappings of each. However, to efficiently compute an actual CP, the program must first find the *essential* difference, if any, between the outputs of the two models. At first sight, the abstract description of a CP for two models whose outputs, respectively, are O_1 and O_2, is: Any input, I, such that $O_1 \neq O_2$. Whilst adequate, such a CP description can lose much of the information contained in the finer-level detail of the two outputs. It describes the requirements of the CP, but does not make reference to any abstract difference between O_1 and O_2 which would enable a CP to be found more quickly. This section describes how PG computes the essential difference(s) between the outputs of two models.

Consider a pair of models which take a numeric input, **?n**, and each produce a numeric output. One model, M_1, produces the output: **(double ?n is (*times ?n 2))**, and the other model, M_2, produces: **(double ?n is (*times ?n ?n))**. For this pair of models, PG's goal is to find an **?n** such that: **(double ?n is (*times ?n 2))≠(double ?n is (*times ?n ?n))**. To do this, the program could choose some value for **?n**, and substitute is for **?n** in the above expression. Evaluating the predicate, ≠, would then reveal whether the value chosen was suitable. This procedure can be repeated until a suitable value for **?n** is found. However, a lot of redundant effort can be avoided by focussing the predicate, ≠, on those aspects of the expression which can differ, based on the value of **?n**. The only part of the two output expressions which can differ is shown in figure 3-8. Thus, the CP can be more efficiently computed by merely using the following test: **(*times ?n ?n)≠(*times ?n 2)**.

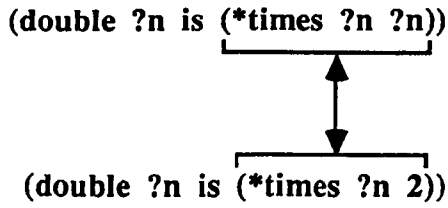


Figure 3-8 — The fundamental difference between the two outputs

The reader may be wondering why we do not delve further into the two function expressions to find a more fundamental difference. The two function expressions are the same apart from the fact that one has the number '2' as its second argument. On the face of it, **?n≠2** is an even more precise description of the required CP (see figure 3-9). However, it is not valid to assume that the overall function expression is different, just because its arguments are. Substituting '0' for '?n' reveals that this is so. The two outputs are equal, despite the fact that **?n≠2: (*times 0 0)=(*times 0 2)**. Nevertheless, for some functions this strategy could form a useful heuristic.

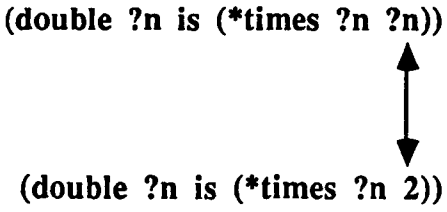


Figure 3-9 — An even more fundamental difference?

To find minimal CP-satisfying tests, such as that in figure 3-8, PG unifies the two output expressions, using the augmented unification algorithm of section 3.5.3. The set of bindings so produced denotes the conditions which must hold for the two expressions to be *equal.*. Thus, the two expressions can only be *unequal* if one or more of those bindings are violated (if there are no bindings, then it is not possible to generate a CP). For example, the two outputs: **(?x ?y)** and **(?y ?x)** can be unified, producing the set of bindings: **{?x/?y}**. Thus, PG's goal should be to find an **?x** and a **?y** such that **?x≠?y**. Similarly, for the two outputs: **(?x ?y ?z)** and **(?a ?b ?z)**, the bindings are: **{?y/?b ?x/?a}**. So, the two expressions unify provided that: **?x=?a ∧ ?y=?b**. Therefore, they do not unify if:

$\neg$(?x=?a $\wedge$?y=?b), i.e. ?x$\neq$?a $\vee$?y$\neq$?b. In this instance, PG's goal would be either to find an ?x and an ?a, such that ?x$\neq$?a, or to find a ?y and a ?b such that ?y$\neq$?b. Either suffices to discriminate between the two models. However, the two abstract outputs: (?x ?y) and (?x ?y) are equal, and so unification produces no bindings. Thus, no CP exists for these two outputs.

In summary, PG computes the CP tests by unifying the two output expressions, turning the bindings into equality constraints, and finally negates the set of equality constraints. This approach suffices for the models used to evaluate PG in Chapter 4. Note that we can handle the commutativity and associativity of functions. For example, given the CP test: (*times ?x ?y)$\neq$(*times ?y ?x), PG would not be doomed to an endless search for an ?x and ?y satisfying this condition, provided that it had the appropriate axioms available. The search an instance which satisfies this constraint would be futile, because multiplication is commutative, i.e. $\forall$(x,y) EQUAL(times(x,y),times(y,x)).

3.10. Summary

In this chapter, we have presented an algorithm for deriving abstract CP descriptions from production rule models by Abstract Interpretations. This technique involves running a pair of production rule models on abstract, rather than concrete, data objects. In the following chapter, we shall evaluate the algorithm using production rule models from the domain of fraction subtraction. The ensuing chapter will also briefly cover the theoretical limitations of the algorithm, and methods of improving the efficiency of the implementation.

Chapter 4

An Evaluation of PG

4.1. Introduction

In order to obtain a feel for the applicability of PG to the problem of I/O mapping derivation, we will evaluate the program on production rule models from the domain of fraction subtraction. We then describe the study from which the fraction subtraction models are taken. PG is tested by comparing its predictions for the 20 models with those found by exhaustively running the models on concrete problems. This is followed by a discussion of the results and some proposals for improving PG's efficiency. First though, we apply PG to the models which stumped RPC in chapter 2.

4.2. The 'Chapter 2' Models Revisited

In Chapter 2, the strengths and weaknesses of RPC were highlighted with the help of a number of production rule models, specially created for the job. Before evaluating PG, it would be prudent to verify that it is able to handle the models which foxed RPC. We shall also use this opportunity to describe the trace information produced by PG.

4.2.1. PG's Runtime Trace Output

With the help of a pair of simple rulesets, we shall now describe how to interpret the trace output of PG. Both rulesets are taken from the rules of the **EVEN-ODD** example, presented in section 2.5.5. The first ruleset, 'EVEN', consists of the rules **r1** and **r3**, whilst the second, 'ODD', comprises **r2** and **r4**, as shown below.

```
Model: EVEN
(define-rule r1
 if (first-feature ?x) & (*evenp ?x)
 then (next-feature ?x) & (*output (?x is even)))

(define-rule r3
 if (next-feature ?x) & (*evenp ?x)
 then (*output (and its successor is odd)) & (*halt))


Model: ODD
(define-rule r2
 if (first-feature ?x) & (*oddp ?x)
 then (next-feature ?x) & (*output (?x is odd)))

(define-rule r4
 if (next-feature ?x) & (*oddp ?x)
 then (*output (and its successor is even)) & (*halt))
```

The ruleset, **EVEN**, accepts an **?x** which is even, outputs **(?x is even)**, then **(and its successor is odd)**, and then halts. Ruleset **ODD** outputs **(?x is odd)**, and **(and its successor is even)**, provided that the initial **?x** is odd. The trace begins by naming the two models to be analysed, listing the Input Specifications, and printing a reminder of how conflict resolution is displayed:

It then shows which Input Specification it has chosen to work on. This is followed by a series of numbered 'frames', each showing a particular cycle of the interpreter. Once it has completed processing one Input Specification, it selects another and begins displaying frames pertaining to that Input Specification. This continues until no Input Specifications are left. If there is an opportunity, within the current frame, to try to generate a CP from the current I/O information, then this is displayed. PG does not actually try to generate a CP, but carries on

until it has completed an exhaustive search. However, the trace shows the information which would have been used to generate a CP, had that part of PG been invoked at that point.

Each frame displays the name of the model being processed, the results of any theorem-proving attempts, conflict resolution, exclusion clause generation, and the results of any attempts to pair the current state with those of the other model. In addition, when a rule fires, the trace shows the new value of Working Memory, the constraint-set, the exclusion clauses, and any new outputs (in the interests of readability, all variables are fully instantiated to level 0, thus one would never see, for example, **?x.5**, but rather its binding which would either be some constant or a level 0 variable, e.g. **?x.0**).

We now present an annotated trace of PG processing the **EVEN** and **ODD** rulesets (annotations are in italics). The axiom, used for this proof is: $\forall(x)\ \neg(\text{EVENP}(x) \wedge \text{ODDP}(x)) \wedge (\text{EVENP}(x) \vee \text{ODDP}(x))$.

The ruleset is given a single Input Specification, consisting of a single Working Memory element: ***(first feature ?z).***

```
CP-search for rulesets EVEN and ODD,
on Input Specs: (((first-feature ?z)))
In the following trace, 'CS' denotes the Conflict Set, and the numbers 1, 2,
and 3, on the same line, denote the application of Refractoriness, Recency, and
Specificity, respectively.
```

PG selects the first Input Specification, and then matches it with the rules in the two models. In the ruleset, EVEN, there is one instantiation of the rule, r1, and it is named: #r1. PG then checks the constraints of the instantiation. 'Satisfiables' are constraints known to be consistent as they are saved from previous cycles; this is initially NIL. The rule, r1, contains a single constraint, ***(*evenp ?x)****, which is passed to the theorem prover, and found to be consistent (note that the '?x.1' is replaced by its binding of '?z.0'. There are no other instantiations in the Conflict Set, thus the competitors are NIL, no exclusion clauses need be generated, and no instantiations are contradictory (i.e. have violated exclusion clauses). Similarly, in frame 2, the instantiation #r2 has no competitors, and therefore no exclusion clauses. Frame 2 finishes with an attempt to form a State Pair from instantiations #r1 and #r2. One of two instantiations is arbitrarily chosen to form the negated theorem (in this instance #r2). The theorem prover finds a refutation of* ***EVENP(?z.0) ∧ ODDP(?z.0)****, and so the pairing is rejected. However, as no consistent pairings are possible, it does create two forced-halt pairings. This completes the initialisation.*

```
_____________________________________________________________________
Choosing Input Specification: ((first-feature ?z.0))

[1]_____________________________________________________________
Context: EVEN
Checking pattern-set consistency for #r1...
  Satisfiables: NIL
  Negated Theorem: EVENP(?z.0)... is consistent.
CS: (#r1), 1: (#r1), 2: (#r1), 3: (#r1)
#r1 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

[2]_____________________________________________________________
Context: ODD
Checking pattern-set consistency for #r2...
  Satisfiables: NIL
  Negated Theorem: ODDP(?z.0)... is consistent.
CS: (#r2), 1: (#r2), 2: (#r2), 3: (#r2)
#r2 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL
Checking consistency of State Pair...
  Satisfiables: EVENP(?z.0)
  Negated Theorem: ODDP(?z.0)... is not consistent.
The pairing: EVEN/#r1 with ODD/#r2 is inconsistent.
EVEN is in a forced halt state.
ODD is in a forced halt state.
(Initialisation Complete)
```

Having completed the initialisation, PG returns to frame 1 and fires the instantiation #r1. The elements deposited, Working Memory, the constraints and exclusions collected so far, and any outputs are displayed in this frame. Note that the element, ***(next-feature ?z.0)****, is actually deposited as:* ***(next-feature ?x.1)****. However, all of the variables in the trace are fully instantiated before printing, as this improves readability.*

```
Selecting context: EVEN

[1]Firing: #r1
Deposited: ((next-feature ?z.0))
WM: ((next-feature ?z.0) (first-feature ?z.0))
Constraints: EVENP(?z.0)
Exclusions: NIL
New outputs: ((*output (?z.0 is even)))
```

PG now enters the next Recognise-act Cycle (frame 3). There are now two instantiations, #r1 and #r3, however, Refractoriness rejects #r1 as it has already fired on a previous cycle. PG tries to pair the new state with that of the model, ODD, but fails because the constraints are incompatible. Note that it does not have to call the theorem prover, because the unsatisfiability of the constraint pair was cached during the previous cycle. The trace also informs us that 'ODD' is in a forced-halt state. This is the same actual state as that referred to in frame 2. This state has now been paired with a new 'EVEN' state, hence the mention in the frame below.

```
[3]_____________________________________________________
Context: EVEN
CS: (#r1 #r3), 1: (#r3), 2: (#r3), 3: (#r3)
#r3 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL
Checking consistency of State Pair...
  Satisfiables: EVENP(?z.0)
  Negated Theorem: ODDP(?z.0) (already known)... is not consistent.
The pairing: EVEN/#r3 with ODD/#r2 is inconsistent.
ODD is in a forced halt state.
```

*The instantiation, #r3, now fires signalling a halt on this path. PG prints out the information which would be used to generate a CP. As described in section 3.8.3, if PG is unable to pair any states at all, then it pairs each state from the pair with a copy of the inconsistent state, where the latter has been forced to halt. This pairing occurred at the end of frame 2. The CP-generation frame shows the original Input Specification, and the information contained in each state of the pair. The ruleset EVEN fired two instantiations, output two elements, and requires that '?z.0' be 'EVENP'. The ruleset, ODD, halted without producing any behaviour. Paraphrasing this frame: Given an input of the form, **(first-feature ?z.0)**, where ?z.0 is even, the ruleset, EVEN, outputs the elements: **(?z.0 is even) (and its successor is odd)**, whilst the ruleset, ODD, outputs nothing at all. The CP-generation frame also tells us that the models can be distinguished by the fact that they produce different numbers of outputs.*

```
[3]Firing: #r3
WM: ((next-feature ?z.0) (first-feature ?z.0))
Constraints: EVENP(?z.0)
Exclusions: NIL
New outputs: ((*output (and its successor is odd)))
Halt signalled on this path.

_____________________________
CP-generation info:
Input Spec: ((first-feature ?z.0))

State information for STATE1. Ruleset: EVEN
 Instantiations: (#r1 #r3)
 Outputs: ((*output (?z.0 is even)) (*output (and its successor is odd)))
  EVENP(?z.0)
Exclusion clauses: NIL
This ruleset is in a halt state.

State information for STATE2. Ruleset: ODD
 Instantiations: NIL
 Outputs: NIL
This ruleset is in a halt state.

The models can be distinguished by the fact that they produce different numbers
of outputs.
Switching rulesets because of *HALT.
```

Because the ruleset, EVEN, has just halted. PG switches attention to the other ruleset. The only other unhalted State Pair left is the one generated in frame 2. Instantiation #r2 fires, followed by #r4 in frame 4

```
Selecting context: ODD

[2]Firing: #r2
Deposited: ((next-feature ?z.0))
WM: ((next-feature ?z.0) (first-feature ?z.0))
Constraints: ODDP(?z.0)
Exclusions: NIL
New outputs: ((*output (?z.0 is odd)))

[4]_____________________________________________________________
Context: ODD
CS: (#r2 #r4), 1: (#r4), 2: (#r4), 3: (#r4)
#r4 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL
Checking consistency of State Pair...
  Satisfiables: EVENP(?z.0)
  Negated Theorem: ODDP(?z.0) (already known)... is not consistent.
The pairing: EVEN/#r1 with ODD/#r4 is inconsistent.
EVEN is in a forced halt state.

[4]Firing: #r4
WM: ((next-feature ?z.0) (first-feature ?z.0))
Constraints: ODDP(?z.0)
Exclusions: NIL
New outputs: ((*output (and its successor is even)))
Halt signalled on this path.
```

*The CP-generation information, below, specifies that the ruleset, ODD, outputs the elements: **(?z.0 is odd) (and its successor is even)**, provided that the '?z.0' in **(first-feature ?z.0)** is odd. Under these conditions, the ruleset 'EVEN' generates no observable behaviour.*

```
CP-generation info:
Input Spec: ((first-feature ?z.0))

State information for STATE1. Ruleset: EVEN
 Instantiations: NIL
 Outputs: NIL
This ruleset is in a halt state.

State information for STATE2. Ruleset: ODD
 Instantiations: (#r2 #r4)
 Outputs: ((*output (and its successor is even)) (*output (?z.0 is odd)))
  ODDP(?z.0)
Exclusion clauses: NIL
This ruleset is in a halt state.

The models can be distinguished by the fact that they produce different numbers
of outputs.
```

As before, PG switches rulesets because the current one has just halted. However, there are no State Pairs left containing the ruleset, EVEN, in an unhalted state, so it tries switching back to 'ODD'. All of the 'ODD' states have halted too, so PG tries to process the next Input Specification. As there was only one Input Specification to begin with, PG ceases processing the pair of models, having completed an exhaustive search.

```
Switching rulesets because of *HALT.

Selecting context: EVEN
Switching ruleset because the current ruleset has no unhalted State Pairs.

Selecting context: ODD

No input specifications left.
```

The traces, included in the appendices, will follow the above format, although they will not be so copiously annotated. The following section presents the CP-generation information produced by PG when applied to the models in Chapter 2.

4.2.2. PG Applied to the Chapter 2 Models

In section 2.5.4, RPC successfully generated the I/O mappings for the rulesets 'SUBTRACT' and 'ABS-SUBTRACT'. PG generated equivalent I/O mappings, as shown below. Note that, because PG analyses the rulesets in pairs, the final CP-generation information is more detailed than that produced by RPC. Whereas RPC just produced a list of the I/O mappings for each ruleset, PG produces a list of pairings, where each pairing represents an overlapping set of inputs. Before generating a problem from RPC's abstract I/O mappings, the program would have to find a pair of mappings (one from each list) where the abstract input specifications overlap. These pairings are automatically produced by PG, as it always Abstractly Interprets the rulesets in pairs (a full trace of this and subsequent runs can be found in Appendix III).

The CP-generation information below, tells us that an unequal pair of output elements is produced for an input of the form: ***(?m.0 - ?s.0)****, where* ***LT(?m.0,?s.0)*** $\wedge$ $\neg$***LT(?s.0,?m.0)***.

```
CP-generation info:
Input Spec: ((?m.0 - ?s.0))

State information for STATE1. Ruleset: SUBTRACT
 Instantiations: (#negative-result #subtract #halt1)
 Outputs: ((*output -) (*output (*subtract ?s.0 ?m.0)))
  LT(?m.0,?s.0)
Exclusion clauses: LT(?m.0,?s.0)
This ruleset is in a halt state.

State information for STATE2. Ruleset: ABS-SUBTRACT
 Instantiations: (#swap-numbers #halt2)
 Outputs: ((*output +) (*output (*subtract ?s.0 ?m.0)))
  (¬LT(?s.0,?m.0) ∧ LT(?m.0,?s.0))
Exclusion clauses: LT(?m.0,?s.0)
This ruleset is in a halt state.

The outputs - and + cannot be unified. Therefore, a problem, satisfying the
above constraint set, will discriminate between the two models.

No CP can be generated for the outputs (*subtract ?s.0 ?m.0) and (*subtract
?s.0 ?m.0), as they are equal.
```

For an input of the form: ***(?m.0 - ?s.0)****, where* ***¬LT(?m.0,?s.0)****, the outputs are always equivalent.*

```
CP-generation info:
Input Spec: ((?m.0 - ?s.0))

State information for STATE1. Ruleset: SUBTRACT
 Instantiations: (#positive-result #subtract #halt1)
 Outputs: ((*output +) (*output (*subtract ?m.0 ?s.0)))
  ¬LT(?m.0,?s.0)
Exclusion clauses: ¬LT(?m.0,?s.0)
This ruleset is in a halt state.

State information for STATE2. Ruleset: ABS-SUBTRACT
 Instantiations: (#positive-result #halt2)
 Outputs: ((*output +) (*output (*subtract ?m.0 ?s.0)))
  ¬LT(?m.0,?s.0)
Exclusion clauses: ¬LT(?m.0,?s.0)
This ruleset is in a halt state.

No CP can be generated for the outputs + and +, as they are equal.

No CP can be generated for the outputs (*subtract ?s.0 ?m.0) and (*subtract
?s.0 ?m.0), as they are equal.
```

Although RPC was able to handle the SUBTRACT and ABS-SUBTRACT rulesets, we found that it could not handle a very simple ruleset: 'SIDE-BRANCH' (section 2.6). This was because the ruleset generated an output from a rule which, in terms of the dependency analysis, was not on a direct path to the halt rule. As we can see from the CP-generation information, below, such side branches are not problematic for PG.

```
CP-generation info:
Input Spec: ((go))

State information for STATE1. Ruleset: SIDE-BRANCH
 Instantiations: (#start #side-branch #halt)
 Outputs: ((*output 1) (*output (*add1 1)))
  NIL
Exclusion clauses: NIL
This ruleset is in a halt state.
```

The final example, presented in section 2.6, was of a pair of rulesets which can only be distinguished by taking Recency into account. RPC was not able to analyse the ONE-TWO1 and ONE-TWO2 rulesets, because it could not reason about the conflict resolution principle: 'Recency'. Again, this causes PG no problems, because it encodes an abstract version of PGPS's conflict resolution strategy.

```
CP-generation info:
Input Spec: ((start))

State information for STATE1. Ruleset: ONE-TWO1
 Instantiations: (#r1 #r2 #r3)
 Outputs: ((*output 1) (*output 2))
  NIL
Exclusion clauses: NIL
This ruleset is in a halt state.

State information for STATE2. Ruleset: ONE-TWO2
 Instantiations: (#r1-swapped #r3)
 Outputs: ((*output 2))
  NIL
Exclusion clauses: NIL
This ruleset is in a halt state.

The models can be distinguished by the fact that they produce different numbers
of outputs.
```

From these examples we can see that PG overcomes the limitations of RPC, highlighted in Chapter 2. In the next section we discuss ways of more generally evaluating PG.

4.2.3. Conclusions

In the previous section, we found that PG is well able to handle the models which caused RPC so much trouble in Chapter 2. We should now satisfy ourselves that PG is able to handle more than just simple, handcrafted examples of this sort.

There are a number of approaches open to us in evaluating PG, indeed, there are a number of questions which could be answered by such an evaluation. Such questions include:

(i) How general is the algorithm? i.e. to what class of models can it be applied?

(ii) How does it perform on empirically-derived models, as opposed to specially-created examples?

(iii) What is the algorithm's time complexity? i.e. what factors affect its performance?

(iv) How well would the algorithm mesh with a 'real' tutoring system?

The remainder of this thesis will be devoted to answering these questions. Question (i) will be dealt with by reasoned argument, in section 4.7. Question (ii) is covered by evaluating PG on a set of empirically-derived models of children solving fraction subtraction problems. These models are a good test of PG because they pertain to a domain which is quite complex (at least when compared with other arithmetical domains such as multi-column subtraction). They are also a fairly objective test, as they were not specially created to test PG, but were pre-existing (however, this statement should be tempered by the fact that, during PG's development, it was always known that these models would be used as a test of its performance). Section 4.8.1 deals with question (iii), and suggests modifications which might improve PG's efficiency. Question (iv) is covered in section 5.3.

In the following section, we introduce the domain to be used in evaluating PG.

4.3. An Empirical Investigation of Fraction Subtraction Errors

As mentioned in the previous section, we shall evaluate PG using a set of empirically-derived production rules, from the domain of fraction subtraction. This section describes how these models were derived. The study described in this section was carried out by the author prior to the commencement of the degree for which this thesis has been submitted. However, the study was seminal in the development of the ideas leading to PG. The following account has been included in the thesis, because it provides important background information about the models used to evaluate PG. The study is more fully described in Evertsz (1982).

4.3.1. The Skill of Fraction Subtraction

The algorithm, taught in British schools, for solving simple fraction problems (i.e. those not involving whole numbers), is as follows. If the denominators are equal, then just subtract the numerators. If the denominators are not equal then calculate the 'Least Common Multiple' (LCM). This is the lowest number into which both denominators will divide, and becomes the new denominator for the problem. For each fraction, calculate the new numerator by dividing the LCM by the denominator, and multiplying this quotient (factor) by the old numerator. Once this is done, the problem consists of two fractions with equal denominators, so subtract the numerators, putting the result over the new denominator. If the resultant fraction requires cancelling, then find the highest common factor of the numerator and denominator, and divide them by it.

If whole numbers are involved, then the situation is more complicated. There are basically two algorithms, taught in British schools, for dealing with such problems (termed 'mixed fraction problems'). One algorithm (the 'vulgarising algorithm') involves turning a mixed fraction into a simple one, by multiplying the whole number by the denominator, and adding the product to the numerator. For example, $2^1/_4$ becomes $((2x4)+1)/_4$, or $^9/_4$. The vulgarising algorithm finishes by unvulgarising the final fraction, i.e. it is turned back into a mixed fraction by the reversal of the vulgarisation procedure[1]. The other algorithm (termed the 'mixed algorithm') keeps the fractions mixed throughout the solution process. The mixed algorithm caters for cases where the fractional part of the minuend (i.e. the first term) is less than that of the subtrahend (i.e. the second term). In such cases, one must borrow a unit from the minuend's whole number, by decrementing it, and adding $^\delta/_\delta$ to the fractional part of the minuend, where δ is the denominator of the minuend.

Finally, both algorithms must handle problems in which either the minuend or subtrahend has no fractional component. Such terms are called 'loose numbers'. The vulgarising algorithm converts the loose number into an equivalent fraction, where the denominator is equal to that of the other fraction. In the case where the subtrahend is loose, the mixed algorithm deals with the problem by treating the subtrahend as a mixed fraction with a null fractional part. If it is the minuend which is loose, then the mixed algorithm borrows a unit from the whole number in much the same way as it does when forced to borrow from a mixed fraction. The borrowed term becomes the fractional part of the minuend.

[1] In the interests of simplicity, the four error types pertaining to the vulgarising algorithm, v1, v2, v3 and u1, have been excluded from this study.

Some examples of these algorithms are presented below.

Simple fractions: $^3/_4 - {}^1/_4 = {}^2/_4 = {}^1/_2$

$^3/_4 - {}^1/_8 = {}^{(2x3)}/_{(2x4)} - {}^{(1x1)}/_{(1x8)} = {}^6/_8 - {}^1/_8 = {}^5/_8$

Vulgarising: $4{}^1/_3 - {}^2/_3 = {}^{((3x4)+1)}/_3 - {}^2/_3 = {}^{13}/_3 - {}^2/_3 = {}^{11}/_3 = 3{}^2/_3$

$3 - {}^1/_8 = {}^{24}/_8 - {}^1/_8 = {}^{23}/_8 = 2{}^7/_8$

Mixed: $4{}^1/_3 - {}^2/_3 = 3{}^{(3+1)}/_3 - {}^2/_3 = 3{}^4/_3 - {}^2/_3 = 3{}^2/_3$

$3 - {}^1/_8 = 2{}^8/_8 - {}^1/_8 = 2{}^7/_8$

4.3.2. The Diagnostic Problem Set

The goal of the study was to develop a production rule account of the errors which children make when solving fraction subtraction problems, and to ascertain whether the errors could be explained without recourse to mal-rules. After an initial pilot study, conducted to ascertain the feasibility of modelling fraction subtraction skill, protocols were collected of 29 children (aged 11 to 16) solving a specially-designed diagnostic problem set. The pilot study revealed a number of deficiencies in the problems given to the children. For example, in all of the problems which involved cancelling, the numerator of the penultimate term was equal to its highest common factor ($^2/_4$; $^2/_8$; $^3/_6$). This meant that one could not tell whether the children were cancelling by computing the highest common factor. An equally valid account explains the cancelling behaviour by positing that the children cancel by dividing the denominator by the numerator to derive the highest common factor. To discriminate between these two hypotheses, one must set a problem where the highest common factor is not equal to the numerator of the penultimate fraction (e.g. $^6/_8$). This encounter with the failings of a carefully-designed problem set, was the first time that the author realised that dynamic problem-generation is crucial to accurate diagnosis.

In an attempt to maximise the information contained in the protocols, the problem set was designed to test for particular subskills (e.g. coping with unequal denominators, and cancelling). Furthermore, to reduce the intrusive effects of faulty number fact determination, the numbers in the problems were constrained to single-digits. The following factors were used to generate the problem set:

1: Magnitude relationship between the two numerators.

2: Magnitude relationship between the two denominators.

3: The LCM of the two denominators is either:
(a) obtained by multiplying the two denominators together, e.g. 4 and 9,
(b) the larger of the two denominators, e.g. 3 and 9,
(c) not one of (a) or (b), e.g. 6 and 9.

4: Either:
(a) the first fraction is mixed,
(b) both terms are mixed,
(c) both terms are simple fractions.

5: A borrow is required for some problems where the minuend is mixed.

6: Some problems, when solved using the vulgarising algorithm, should involve unvulgarising the final fraction.

7: The highest common factor for problems involving cancelling should either:
(a) be computable by merely dividing the denominator by the numerator,
(b) not be computable by this method.

8: Some problems should have either a loose minuend, or a loose subtrahend.

The problem set was designed so that each factor could be found in at least two problems. This allowed us to use one problem from each pair to generate the production rule models (termed the 'model set'), and the other to test the degree to which the model correctly predicts the child's answers (the 'test set'). Intermingling the factors, in such a way as to ensure that each was present in at least two problems, yielded 22 problems in all.

4.3.3. Error Classification

Overall, 59% of the problems were solved incorrectly. These errors can be classified according to whether they are number fact, systematic, or unaccounted-for errors. 68% of the errors were adjudged to be systematic, 14% were classified as number fact errors, and 18% were unaccounted for by the production rule models. In this thesis, the models have been translated from their original OPS2 representation to PGPS. They can be found in Appendix IVa, together with the actual models (expressed as lists of production rule names) generated from the data (Appendix IVb). The systematic errors were classified into 'error types'. These are reproduced in table 4-1. Illustrative examples can be found in table 4-2.

Type	Error Description	Delete Rule(s)	Insert Rule(s)	Frequency
w1	The child gives-up with problems where the first whole number is loose.	WN1		10
w2	Where the first whole number is loose, the child writes down the fractional part of the second fraction.	WN1	WN1m	5
w3	Gives-up where the second parameter is a loose whole number.	WN2		1
w4	Cannot deal with mixed fraction problems where only the first parameter is mixed.	WN3		1
w5	If after some other mistake the first whole number is less than the second, then take the absolute difference	WN4	WN4m	1
n1	Does not borrow.	NM1		1
n2	Takes the absolute difference between the two numerators.	NM1, NM2ab	NM2m1	3
n3	Takes the absolute difference between the two numerators before dealing with the denominators.	NM1, NM2ab	NM2m2	6
n4	Subtracts numerators before denominators.	NM2ab	NM2m3ab	1
n5	When borrowing, adds ten to the numerator.	BRW	BRWm	2

Table 4-1 – Error types (from Evertsz, 1982)

Type	Error Description	Delete Rule(s)	Insert Rule(s)	Frequency
d1	Cannot calculate the LCM.	DN2		1
d2	If the denominators are not equal, then take their absolute difference.	DN2	DN2m1	7
d3	Subtract the two denominators.	DN1, DN2	DN2m2	1
f1	Works out an LCM, but does not enlarge the numerators.	FCTabcd		1
f2	Calculates correct LCM, but enlarges numerators by cross multiplying.	FCTabcd	FCTvabc	2
f3	Factors, even when the two denominators are equal.	DN1, FCTabcd	DN1v FCTvabc	1
h1	Does not cancel.	HCF		18
e1	Does not change 0/D to zero, (where D is the denominator).	END3		1
e2	Changes 0/D to N/D (where N is equal to the numerators).	END3	END3m1	7
e3	Changes 0/D to D.	END3, END4	END3m2	3
e4	As e2, but changes a whole number from 0 to W (where W is equal to the two whole numbers).	END4	END4m1	3
e5	Leaves the whole number out of the answer.	END2		1

Table 4-1 (continued) - Error types (from Evertsz, 1982)

An example of each of these error types is presented in table 4-2, below. Asterisks are used to represent unspecified parts of the problem solving, i.e. those parts which are independent of the error being illustrated. An asterisk can be replaced by various digits, depending on what other errors are present in the student's algorithm.

Type	Error Example
w1	$2 - {}^{3}/_{4} = stuck$
w2	$4 - 2^{7}/_{8} = 2^{7}/_{8}$
w3	$3^{1}/_{2} - 2 = stuck$
w4	$4^{1}/_{4} - {}^{3}/_{4} = stuck$
w5	$\ldots 2^{3}/_{4} - 4^{1}/_{4} = 2^{(3-1)}/_{4}$
n1	$2^{1}/_{4} - 1^{3}/_{4} = 2\ stuck$
n2	$2^{1}/_{4} - 1^{3}/_{4} = 1^{2}/_{4}$
n3	$2^{1}/_{6} - 1^{2}/_{3} = 1^{1}/_{*}$
n4	${}^{3}/_{4} - {}^{1}/_{3} = {}^{2}/_{*}$
n5	$3^{1}/_{8} - {}^{3}/_{8} = 2^{11}/_{8} - {}^{3}/_{8}$
d1	${}^{3}/_{4} - {}^{1}/_{3} = stuck$
d2	${}^{3}/_{7} - {}^{1}/_{9} = {}^{*}/_{2}$
d3	${}^{3}/_{4} - {}^{1}/_{3} = {}^{*}/_{1}$
f1	${}^{5}/_{6} - {}^{1}/_{3} = {}^{5}/_{6} - {}^{1}/_{6}$
f2	${}^{1}/_{4} - {}^{1}/_{6} = {}^{6}/_{12} - {}^{4}/_{12}$
f3	${}^{7}/_{8} - {}^{1}/_{8} = {}^{56}/_{8} - {}^{8}/_{8}$
h1	${}^{3}/_{4} - {}^{1}/_{4} = {}^{2}/_{4}$
e1	${}^{4}/_{7} - {}^{4}/_{7} = {}^{0}/_{7}$
e2	${}^{4}/_{7} - {}^{4}/_{7} = {}^{0}/_{7} = {}^{4}/_{7}$
e3	${}^{4}/_{7} - {}^{4}/_{7} = {}^{0}/_{7} = 7$
e4	$1^{1}/_{2} - 1^{1}/_{2} = 0^{*}/_{*} = 1^{*}/_{*}$
e5	$2^{7}/_{9} - {}^{1}/_{9}\ {}^{6}/_{9}$

Table 4-2 – Error examples

The predictive accuracy of the models is quite high, as shown in table 4-3. This tells us that the models are quite successful (at least 86%) in capturing whatever regularity there is in the data. We say that the models are *at least* 86% successful because, in the worst case, where the

children are 100% consistent, our models concur 86% of the time with the data. If we allow for slips of action on the part of the children, then the coverage of their systematic behaviour is even better. Unfortunately, this tells us very little about the models' psychological validity. From a computational perspective, fraction subtraction is a fairly easy skill to capture in production rule format, thus, to the extent that the children's behaviour is systematic, it should come as no surprise that we have been able to model this systematicity.

Nevertheless, there is other, more qualitative evidence, in favour of the models' validity. There is considerable overlap between the protocols of the children, and those produced by the production rule models. For a given child/model pair, the following is true:

(i) the model and child deal with the various parts of the problem in the same order.

(ii) the model deposits the same intermediate numbers in Working Memory, as the child writes on the paper.

(iii) the model makes use of very few hypothesised internal signals, i.e. most of the information used by the model can be found in some form or other in the child's protocol.

(iv) where the child's bug is accounted-for by a rule deletion, causing the child to get stuck, both child and model are found to halt in the same place.

Model Set	Test Set	Overall
89%	86%	87%

Table 4-3 – Mean proportion of answers accounted for.

We shall now discuss how these models are to be used in evaluating PG.

4.4. Choosing Fractions Models to Evaluate PG

How should we use the fractions models to evaluate PG? At first sight, one good method would be to randomly select pairs of models, and pass them to PG. However, such pairings

would bear no relation to those which would be proposed by a typical tutoring system. Each of the fractions models covers all of the 22 problems in the original problem set. These problems are of many different types (the factors, incorporated in the problems, were outlined in section 4.3.2). Some parts of a given model will be for dealing with whole numbers, others for problems where the denominators are unequal, and yet others for problems where a borrow needs to be performed. If the two models of a given pair differ along a number of these dimensions, then PG will produce a number of different CP-generation frames. However, this was not how we had envisaged PG being used. PG is intended to test specific hypotheses about a student's earlier performance. For example, if there are two ways of accounting for how the student dealt with the whole numbers, then those two accounts form the hypotheses which PG should work on. The models in Appendix IVb cover many different classes of problem, and are really a conglomeration of hypotheses about different aspects of fraction subtraction performance. Therefore, as they stand, these models are not well suited to evaluating PG.

Ideally, we should pass PG two models whose outputs overlap for some class of inputs, and see whether it generates a CP description which includes a non-overlapping set of outputs. Such a test is analogous to the situation where there are two possible accounts for the student's answers so far, and the tutoring system is attempting to generate an input for which the two models produce differing outputs. Generating such models by hand is not an easy task. Because we do not have access to a student modelling system which could automate this model-generation process, we must find a compromise solution. This solution should produce models containing a single error, and should allow us to generate the required models by hand.

With this need in mind, we have chosen to use the error types of table 4-1 as a basis for generating the models. Using the corpus of rules in Appendix IVa, we will generate, for each error, a model containing sufficient rules to produce that error and no more. For each such error model, a companion, bug-free version will be produced - one which solves the same class of problems, without error. In order to run PG on such an error/correct model pair, we need to provide it with an abstract Input Specification, whose set of concrete instantiations includes those for which the buggy model produces an error. Of course, we could have passed PG the whole set of Input Specifications for fraction subtraction. However, by manually choosing the appropriate one, we are giving it the helping hand it would receive were it part of a tutoring system. In a well-designed tutoring system, PG would not only be

passed a pair of models whose outputs overlap for some subset of the earlier problems, but would be passed the Input Specification which was used to generate those earlier problems. For example, if there were some confusion as to how the student deals with problems where the minuend is a loose whole number, then an Input Specification describing such a class of problems would be the one to be used to generate a CP.

Appendix IVb lists the models used to evaluate PG. Through its name, each error model can be related to the error type from which it was derived. For example, the model **error-n2** is the error model for the error type **n2**. Some models incorporate more than one error; such models were created to cover instances where one error type depends for its existence on another. Each 'correct' model typically covers several error models. This is because we only need one correct way of dealing with a particular problem characteristic to cover all of the error models for that problem feature. Similarly, one correct model can be used to cover several orthogonal aspects of error performance (e.g. a model which correctly handles numerators *and* denominators can be used to cover all of the error models pertaining to those two features).

4.5. The Success/Failure Criterion

Having chosen the data to be used in evaluating PG, we now need to define a criterion with which to assess PG's success with that data. This criterion was set out in section 3.2 where we said that PG's search for CPs should be both *complete* and *sound*. 'Completeness' entails being able to find a CP if one exists, whereas 'soundness' means that PG should not generate non-discriminatory problems. The reader will recall that in sections 3.8.2 and 3.8.3, we described a limitation which was deliberately incorporated in PG to reduce search. Although this limitation is just an implementation detail (we described how to overcome it) and does not violate the criterion of completeness, it does preclude exhaustiveness: because PG ignores non-intersecting State Pairs (unless there are no intersecting ones at all), CPs pertaining to those State Pairs will not be found. Although this restriction means that PG will not always find *all* CPs, it does not prevent PG from finding at least one CP if one exists. This was why we allowed the restriction in the algorithm; PG is not required to find all solutions - in a tutoring context it would suffice to find just one CP to present to the student. Thus, although under certain conditions, we do expect PG to fail to find all solutions, we have decided to use exhaustiveness as an evaluation criterion. In our evaluation, it would be useful to know

whether PG manages to generate all possible CPs for each model pair. On the basis of the analysis in section 3.8.2, we expect that PG will only fail to find all CPs in cases where the non-intersecting State Pair restriction is invoked.

Although prepared to relax the exhaustiveness criterion, we are not willing to relax the soundness criterion. If PG generates any problems which are non-discriminating, then that will be deemed a failure of the algorithm.

We now need to decide on the mechanics of the evaluation. The evaluation procedure must enable us to assess PG's success from the exhaustiveness and soundness viewpoints. This can be achieved by exhaustively running the models on the whole set of concrete inputs which they can process, yielding a set of concrete input/output pairs for each model. We can then take each pair of models (correct vs. error model) and extract the complement of the intersection of the two sets of input/output pairs (i.e. the set of input/output pairs which is in either of the two models but not both). This is the set of CPs (each coupled with its concrete output) for the model pair and represents, if you like, the empirical data to be used in the evaluation. If we now take PG's abstract input/output mappings (for each model pair) and instantiate them, we will obtain the set of CPs predicted by PG. On the basis of the empirical evaluation, PG will be dubbed 'sound' if all of its predictions are supported by the empirical data, 'complete' if at least one of its predictions is true, and 'exhaustive' if its predictions cover all of the empirical data. Note that the first and last criteria are not the same, even though they look quite similar. Soundness is satisfied iff PG's predictions are a *subset* of the empirical data; exhaustiveness is achieved iff the set of PG's predictions is *equal* to the set of empirical data. Finally, even if successful, this evaluation does not constitute *proof* of PG's soundness and completeness, it merely provides empirical support for the conjecture; such a proof would be a major undertaking in itself.

The above formulation suffers from one logistical drawback: the production rule models can handle an infinite set of problems. Therefore, we need to restrict the range of concrete values which are used to instantiate the problem descriptions. In the original study (Evertsz, 1982) the models were designed to handle fraction problems where all of the components were single-digit integers, greater than zero (i.e. in the range 1 to 9). We will adopt a similar restriction when instantiating the problem descriptions and constrain the problems to only consist of integers between 1 and 5. This will not invalidate the evaluation, because none of the models depend on the presence of integers greater than 5.

4.6. The Evaluation

4.6.1. Introduction

All of the data pertaining to this evaluation can be found in Appendices IVa - IVe. The PGPS fraction rules are in Appendix IVa. A list of the correct and error models can be found in Appendix IVb. Appendix IVc contains the CP information generated by PG for each model pair, and also includes the axioms supplied to PG's theorem prover. We have not provided full traces for all of the model pairs, because they are almost twice as long as the whole thesis itself, and are not very interesting. The critical part of the traces is the final output of each run provided in Appendix IVc. However, in Appendix IVe, we have provided a full trace of PG running on the error type H1. Appendix IVd shows the results of the evaluation; for each model pair, this appendix shows the CPs generated, the corresponding outputs and any erroneous predictions discovered.

The empirical data for the models was generated by running each model on all possible concrete inputs which satisfied the Input Specification. Similarly, PG's predictions were obtained by instantiating the input/output mappings with all possible values (between 1 and 5) which satisfied the CP descriptions. For example, PG produced the following CP description for the model pair CORRECT-W1&2 vs. ERROR-W2 (taken from Appendix IVc):

CP-generation info:
Input Spec: (((WN 1) ?WX.0) ((WN 2) ?WY.0) ((FR 2) ?NY.0 / ?DY.0))
The outputs ((*SUBTRACT (*DEC ?WX.0) ?WY.0) AND (*SUBTRACT ?DY.0 ?NY.0) / ?DY.0)
and ((*SUBTRACT ?WX.0 ?WY.0) AND ?NY.0 / ?DY.0)
can be unified. Therefore, a CP could be generated provided the following constraint-set is instantiable:
((¬EQ(?NY.0,SUB(?DY.0,?NY.0)) ∨ ¬EQ(SUB(DEC(?WX.0),?WY.0),SUB(?WX.0,?WY.0)))
∧ ¬EQ(?NY.0,0) ∧ ¬EQ(SUB(?WX.0,?WY.0),0) ∧ ¬EQ(SUB(DEC(?WX.0),?WY.0),0)
∧ ¬EQ(SUB(?DY.0,?NY.0),0) ∧ LT(?WY.0,?WX.0) ∧ LT(?NY.0,?DY.0))

To generate the concrete CPs in the above example, PG instantiated the variables **?WX, ?WY, ?NY** and **?DY** with all permutations of the integers between 1 and 5 inclusive, rejecting those instantiations which did not satisfy the above constraint expression. Note that

the search is quite inefficient, because it is exhaustive[1], in fact, it is about as naive a constraint-satisfaction algorithm as one could conceive of. Very little effort was devoted to the problem of instantiating PG's CP descriptions. We could certainly have implemented a more efficient constraint-satisfaction algorithm based on the large body of work in this area; but we chose to concentrate on the problem of analysing the student models into expressions suitable for processing by a constraint-satisfaction system. In section 4.8.3, we discuss solutions to the problem of efficient constraint satisfaction.

The results of the comparison between PG's predictions and the empirical data generated by PGPS are presented in the next section.

4.6.2. Results

In the following table, the first column shows the Error Type, the second the number of CPs predicted by PG, the third the number found by exhaustively running PGPS and the fourth lists the number of legal problems (based on the Input Specification) for each error type.

[1].The instantiation process was implemented as follows: Each variable in the input specification is represented as a number-generating closure (i.e. a stream of integers). Each instantiation is generated by first asking each integer stream to generate its next value, and then evaluating the constraint expression with respect to the current set of variable bindings. If the instantiation passes this test, then it is added to the set of CPs. The method of testing the constraint expression is quite efficient. A naive solution would be to instantiate the variables therein, and then evaluate the expression. A more efficient solution is to invoke the Lisp compiler on the expression to obtain a function object. This compiled function can then be applied to a list of the variable bindings. In Procyon Common Lisp v2.1.4 (on an Apple Macintosh SE/30), this solution is 50 times faster than the former. Thus, the expression below is solved in 0.58s instead of 29s. There are 729 possible instantiations of the 3 variables below (each variable ranging between 1 and 9), however, only 36 of them are valid:
(and (not (= (- x y) 0)) (< y x) (= (1- u) 0) (not (= x y))).

Type	PG Predicted	PGPS Found	Maximum Possible
w1	45	45	45
w2	90	90	90
w3	135	135	135
w4	40	40	405
w5	48	48	1215
n1	80	80	280
n2	80	80	80
n3	2	2	2
n4	14	14	14
n5	80	80	80

Table 4-4 – PG/PGPS Correspondence

Type	PG Predicted	PGPS Found	Maximum Possible
d1	28	28	28
f1	28	28	28
f2	2	2	28
f3	17	17	17
h1	1	1	25
e1	9	9	25
e2	9	9	25
e3	9	9	25
e4	85	85	125
e5	80	80	250

Table 4-4 (continued) - PG/PGPS Correspondence

From Table 4-4, we can see that PG was successful in predicting the range of CPs for all of the model pairs. Thus, 'soundness', 'completeness' and 'exhaustiveness' were satisfied by this evaluation - PG found all of the 882 possible CPs across the 20 model pairs.

It is interesting that, for some model pairs, all of the possible problems were also CPs (error types: W1, W2, W3, N2, N3, N4, N5, D1, F1, F3), whereas for others the percentage was as low as 4% (e.g. error type H1). Unfortunately, these figures tell us nothing about the proportion of CPs which would crop up in a real tutoring context - that would require a separate study. For those interested, a full trace of PG's analysis of H1 can be found in Appendix IVe.

4.6.3. Discussion

Though 100% successful in this evaluation, PG does have its limitations. We know already (section 3.8.3) that under certain conditions the current implementation of PG sacrifices exhaustiveness. This limitation was not revealed by the evaluation.

The method of instantiating derived problem descriptions is rather poor - the instantiation algorithm requires that limits be placed on the range of integers which can be used to instantiate the abstract CPs. One could overcome this restriction by adopting a breadth-first search strategy, but this too would not be particularly efficient.

PG's theorem prover, though adequate, is too slow to be used in a real tutoring context. The original motivation for implementing a theorem prover was to save time. During early analyses of PG's performance on various models, the theorem proving was done by hand - at each theorem proving juncture PG would ask the author to assess whether a given set of clauses was consistent. This was often very difficult and would lead to mistakes which required restarting the analysis from the beginning. The question of theorem proving was originally seen as quite separate from PG's algorithm and not in need of implementation (there are many theorem provers in existence which can do the job far better than anything which could be implemented during the term of this research). However, it became clear that it would be worth investing the three or so weeks required to implement a rudimentary theorem prover, as this would save time in analysing and debugging PG. With the addition of the constraint simplification mechanism (section 3.6.3) and the equality axioms much of the theorem proving work was automated. However, PG still generated more complex theorems which required the support of a larger set of axioms. When these were added, the theorem prover's performance degraded to the point where one could spot inconsistent clause-sets much faster than the machine (typically a factor of 10 more quickly). The problem lay in the fact that the theorem prover's search strategy was fairly unsophisticated. The set of support strategy helped, but the search was still fairly unguided. Analysis of the theorem proving traces revealed that the program would embark down paths which one could instantly see were fruitless. This was mainly due to domain-specific control knowledge which the machine did not have access to.

Therefore, the more complex axioms were left out. A time limit was also added to the theorem prover - if it could not prove the theorem within 10 seconds, then it asked the user to do so. Experience with the theorem prover revealed that it could prove trivial theorems quickly, but if it hadn't succeeded after 10 seconds, then one stood a better chance of doing so by hand. These machine/user interactions can be seen in the trace of error type H1 in Appendix IVe.

The issue of theorem proving efficiency and completeness is discussed further in section 4.8.2.

4.7. PG's Limitations

Though very successful with the fraction subtraction models, PG does embody one fundamental limitation which is not revealed by this evaluation - it cannot reason about the behaviour of models which contain loops. This is not as serious as it sounds because many domains involve no iteration at all.

As an example of a simple loop, consider the simple two-rule ruleset below. This ruleset is triggered by a pattern of the form **(count from ?x to ?y)** and just outputs the numbers between **?x** and **?y** inclusive.

```
(define-rule start-counting
  if (count from ?x to ?y) & ~(number ?x)
  then (number ?x) & (*output ?x))

(define-rule count
  if (count from ?x to ?y) & (number ?previous) & ~(number ?y)
  then (number (*add1 ?previous)) & (*output (*add1 ?previous)))
```

When run on the input **(count from 2 to 6)**, PGPS produces the following behaviour:

```
Final WM:    ((NUMBER 6) (NUMBER 5) (NUMBER 4) (NUMBER 3) (NUMBER 2)
              (COUNT FROM 2 TO 6))

Outputs:  (*OUTPUT 2) (*OUTPUT 3) (*OUTPUT 4) (*OUTPUT 5) (*OUTPUT 6))
```

If we pass this ruleset to PG for analysis, it will fail because of the loop in the rule **count**. On each iteration, **count** takes the element which it deposited on the previous cycle and deposits another isomorphic element with the number therein incremented by one. This process terminates when the deposited number is equal to **?y**. PG's analysis contains some of

this information, but misses out one crucial aspect: the point at which the loop terminates. For the input **(count from ?a to ?b)**, PG's output is as shown below; as far as PG is concerned, this ruleset is non-terminating.

```
WM:      (... (number (*add1 (*add1 (*add1 ?a.0)))) (number (*add1 (*add1 ?a.0)))
             (number (*add1 ?a.0)) (number ?a.0) (count from ?a.0 to ?b.0))

Outputs: ((*output ?a.0) (*output (*add1 ?a.0)) (*output (*add1 (*add1 ?a.0)))
          (*output (*add1 (*add1 (*add1 ?a.0)))) ...)
```

Identifying loop termination is one fundamental problem which PG faces when tackling iterative groups of rules. Other related problems are that of identifying a group of rules which forms a loop and, given that the loop's termination point can be found, characterising the abstract behaviour of the loop in such a way as to allow that description to be manipulated by other rules in the ruleset. As we shall see in section 5.4, PG does not need to *prove* that a loop terminates - it suffices to identify the termination condition.

Deriving an abstract characterisation of loops is of central importance; loops do not exist in isolation, rather, the sequence of WMEs which they produce will form the inputs to other rules in the ruleset. We have not solved these problems, but in section 5.4 we lay some of the groundwork for extensions which would enable PG to analyse iterative PS models.

4.8. Efficiency Issues

4.8.1. Optimising the Abstract Interpreter

PG represents a fairly literal implementation of the algorithm described in Chapter 3. It implements the algorithm as described, without any attempt to improve efficiency by caching information or by using indexing mechanisms. For example, PG took 6.5 minutes to exhaustively process the H1 example. Even when running on concrete data PG is extremely slow; for example, it took 71 seconds to solve the problem $^3/_4 - ^1/_4$ (timings are for Procyon Common Lisp v2.1.4 on an Apple Macintosh SE/30).

PG implements the behaviour of PSs, albeit ones which process abstract data. Therefore, one might expect that the techniques used to accelerate standard PSs would be of use to PG. There are a number of well-known methods of improving the efficiency of PS interpreters, the most widely used being Forgy's 'Rete' match algorithm (Forgy, 1982).

The Rete match algorithm achieves its speed on the basis of one fundamental insight: naive PS interpreters repeat a lot of work on each cycle. For example, if on one cycle the PS instantiates a rule on the basis of some subset of Working Memory, then there is no need to recompute that instantiation on the next cycle; it will still be valid, provided that none of the WMEs to which it refers have been deleted. In most rulesets, the same condition elements can be found across many rules. It is a waste of effort to match the duplicate pattern once for every rule in which it appears - it suffices to match it just once, and to propagate the variable bindings to all of the rules which contain that pattern (note that two patterns can be equivalent even if the variables therein have different names). This matching process occurs as soon as a new element is added to Working Memory. All of the matching patterns become activated by the new addition, and any rules with a newly-complete set of activated patterns become instantiated. This regimen means that rules do not have to be matched against the whole of Working Memory, rather, the interpreter holds partial instantiations which gradually get augmented as Working Memory grows, only being added to the Conflict Set when fully instantiated.

The Rete match algorithm also compiles-out the pattern-matching process which one expects PS interpreters to follow. It does not employ the unsophisticated match process which we described in section 3.5.3, where each element of a condition is matched against each element of a WME, binding variables to constants where necessary. Instead the pattern is compiled into a set of low-level tests on the WME. For example, the following LHS would compile into the sort of code paraphrased below it.

```
LHS: (?x isa ?y) & (?y isa ?z)

Matches if  both WMEs are of length 3,
            the second element of each is 'ISA',
            and  the third element of the first pattern
                 is equal to the first element of the second.
```

The above example conveys some of the flavour of why the output of the Rete pattern match compiler is so efficient. The question is, can we apply similar optimisations to PG?

The caching of instantiations from cycle to cycle is certainly feasible. To enable the use of a flexible control strategy (e.g. heuristic), one would have to augment the Rete match algorithm with a context-switching mechanism. This would allow PG to jump around the search space, saving the old context when it jumps to another part of the space. To achieve this one need only augment PG's current state information with a structure which represents the links between the saved WM state and the LHS condition elements.

We can also propagate the results of unifying a given pattern with WM to all isomorphic patterns. To achieve this, PG's rule compiler would have to normalise the variables in the rules, so that equivalent patterns were obviously so. For example, the two patterns **(?x likes ?y)** and **(?a likes ?b)** would both become **(?1 likes ?2)**. These normalised variables contain a numeric component; this would allow the use of a much more efficient binding mechanism than PG's current 'association list' method. The numeric components could be used to compute *offsets* in the binding environment, enabling very efficient variable lookup (this is a common technique used in Prolog; cf. Warren, 1977).

Finally, the unification process could be compiled into a more efficient set of tests, in the manner of the Rete match algorithm. The above example would now look as follows:

LHS: (?x isa ?y) & (?y isa ?z)

Matches if both WMEs are of length 3,
the second element of each is 'ISA',
and the third element of WME1
can be unified with the first element of WME2.

With these modifications, PG's time complexity would be of the same order of magnitude as that of the Rete algorithm, i.e. $O(\log_2(P))$. This compares very favourably with PG's current performance which, for a given path, is O(P).

In conclusion, it is clear that a literal implementation of the algorithm described in Chapter 3 would not be a practical proposition in a tutoring system. However, large improvements are possible (from a current O(P) to an expected $O(\log_2(P))$). It seems likely that such improvements would enable PG to be used in a practical tutoring system even on today's hardware.

‖4.8.2. Options for Accelerating the Theorem Prover

Theorem proving forms a major part of PG's processing. In this section, we look at ways of improving its efficiency, both by using faster but still complete theorem-proving mechanisms, and incomplete but nevertheless adequate ones.

Our theorem prover is not as slow as it could have been, but clausal resolution theorem provers are slow in general. One reason for this is their use of clause form. Converting formulae to clause form leads to considerable redundancy. For example, **A≡B** becomes **(¬A ∨ B) ∧ (A ∨ ¬B)** in clause form, thus there are now two instances of **A** and **B**. In the worst case, the number of clauses is an exponential function of the size of the formula.

Another problem with clause form is that it loses much of the pragmatic information contained in the original formula. For instance, the formula **A≡B** can be read as "if A is true then so is B; if B is true then so is A". Which, from a deductive point of view, is more focussed than its clause form version. Clause form is such a 'flat' representation that it provides little guidance on which routes to follow through the search space. For example, PG's theorem prover wastes a lot of time resolving the equality axioms with clauses which to the human observer are clearly irrelevant.

Because of these drawbacks, some researchers have proposed non-clausal resolution theorem provers as an alternative (e.g. Stickel, 1982). These can lead to shorter, more natural-looking proofs, but it is not yet clear if either method is more efficient in *all* cases.

An alternative goal to that of developing faster 'complete' theorem provers is to sacrifice completeness in the interests of efficiency. For example, one could use Prolog as a theorem prover - it would not be complete, but it would be very fast. The control structure would need to be modified slightly to avoid the non-terminating deductions which can result from Prolog's unbounded depth-first strategy.

But what would be the effects of using an incomplete theorem prover in PG? The effects are not as damaging as one might suppose. Recall that the theorem prover's purpose is to spot inconsistent paths, thereby enabling PG to prune them from the search space. If, as a result of its incompleteness, a theorem prover fails to reject a set of clauses, then PG will carry on

expanding that path. This path may ultimately yield an output, in which case PG will build a description of its I/O mapping. When paired with an I/O mapping from a competing model, the CP description will be unsatisfiable (because it contains an inconsistent clause set). Therefore, PG will try but will be unable to instantiate the expression.

Therein lies the trade-off; if PG takes short cuts during theorem proving, then it may pay for this in time wasted trying to instantiate the CP description. If it is thorough in its theorem proving, then any CP descriptions will be instantiable, but the time spent theorem proving may well outweigh the time saved when trying to instantiate the CP description.

4.8.3. Is PG better than exhaustive runs of PGPS?

Having developed an algorithm which can generate CPs, it is now time to consider whether this is any better than exhaustively running the student models on concrete data. There are two separate issues here:

(i) given that we can implement an efficient version of both PG and PGPS, would the former plus constraint satisfaction (henceforth termed 'PG+CS') execute more quickly than the latter?

(ii) could PG provide a tutoring system with information which could not be derived from exhaustive runs of PGPS?

The first question is difficult to assess, given that PG and PGPS are so inefficiently implemented. In the H1 example (section 4.8.1) PGPS solved the problem $^3/_4 - {}^1/_4$ in 0.18 of the time taken by PG to search all of the paths through the ruleset. Given that only 1 in 25 of the possible problems were CPs, and assuming a random selection by PGPS, one would expect it on average to find a CP after 12.5 attempts. Thus, on this example PG would have the edge. However, one can extrapolate little if anything from a single example based on a very inefficient implementation.

The advantage which PG has over PGPS is that it is functioning at a higher level. Any improvements made in the abstract domain will transfer to *all* of the concrete instances which are subsumed by that domain. We will now discuss three such measures: (i) using heuristics

to guide PG's search, (ii) using more efficient constraint satisfaction techniques, and (iii) caching results of earlier abstract interpretations.

Guiding PG's Search

One way to improve PG's performance would be to guide its CP search down paths which are more likely to be fruitful. This will be easiest in cases where the number of differences between the two models is small. In such cases, the search could focus on those differences; if there are too many such differences, then little would be gained by concentrating on them.

To illustrate, consider the case where two models are identical except for the fact that one contains an extra rule. Clearly, any difference between the behaviour of the two models will be due to that extra rule. Therefore, it makes sense to guide abstract interpretation down a path which passes through that rule; the other paths are guaranteed not to yield a CP. To be of any use, this heuristic would have to be cheap to compute; in other words it would have to be quicker than the time taken to explore the excluded paths.

To guide PG towards the extra rule we will need a method of estimating which rules lie on a path towards that rule. It is not easy to see how one could glean such information without actually searching the paths in their entirety. Fortunately, we do have a method of doing this - RPC. RPC is poor at finding CPs, but it is good at revealing the dependencies between rules and is certainly much faster than performing full abstract interpretation. By searching from the extra rule *back* towards a start rule (i.e. one satisfied by the Input Specification), RPC would collect those rules which are *likely* to be on the right path. This information could then be used to guide PG's search, although because this a heuristic, there would still be cases where PG has to search the whole space - heuristics by their very nature are not perfect and therefore can be wrong-footed.

As mentioned above, it is only worth employing this strategy when the two models are very similar to one another. It is interesting that such cases are likely to cause PGPS a lot of trouble. If the two models are very similar, then it is likely that PGPS will have to try many combinations before finally finding a CP. In contrast, a guided PG will find such cases easier. This is one sense in which PG+CS is more *powerful* than PGPS. We would expect the tutoring system to only call on PG when it is having trouble discriminating between two hypotheses. This would happen in cases where there is much overlap between the two because they both account for the student's earlier behaviour. Thus, PG's power would

increase in exactly those cases which are important to a tutoring system which is trying to discriminate between two similar, but slightly different hypotheses.

Efficient Constraint Satisfaction

An area where major improvements could be made is that of constraint satisfaction; this is another area where PG can be speeded up in a way which is not possible with PGPS. There has been much research in the area of constraint satisfaction, and much of it could be borrowed wholesale for use in PG; in particular, the work on propagating interval labels seems particularly relevant to domains such as fraction subtraction (Davis (1987) presents a detailed review of such work). In label inference, each 'node' is labelled with a representation of the set of possible values which it can take. As constraint propagation proceeds, these labels get refined until a solution is found. Each node would represent a variable in PG's derived CP description.

A useful heuristic which one could use when instantiating CP descriptions would be to concentrate on the more restrictive constraints. This would reduce the sets of values which must be processed by the other less restrictive ones.

Another approach which would be of use in cases where there is only a small difference between the two models is to concentrate on those constraints which derived from that small difference. A good example of this is our H1 error type. The only difference between the correct and error models is that the latter has the rule **HCF** missing. This rule is responsible for cancelling terms in the final result, where required. This rule was responsible for the constraint, **NEEDS-CANCELLING-P(SUB(?nx,?ny),?d)**, in the final CP description. By recording where the constraints come from, we could have PG concentrate on those which were derived from the fundamental difference between the two models.

Such optimisations are not possible with PGPS because it does not represent the models in a suitable form. PG's CP descriptions, on the other hand, are in exactly the right form for applying constraint satisfaction techniques.

Caching the Results of Abstract Interpretations

It should be possible for PG to improve its performance by storing the results of earlier model analyses. For example, the I/O mappings for a model could be re-used for comparison

with others - it is not necessary to re-derive them every time the model is being considered as a hypothesis. Another possibility is to compile groups of rules into 'macro-rules' which just describe the I/O mapping of the group of rules. These macro-rules could then be substituted for the original group of rules, leading to models which are much easier to analyse. This is discussed further in section 5.6.1 with regard to its use in optimising rulesets.

Summary

We do not have enough information in order to make a firm decision on whether PG+CS or PGPS is the faster. However, there are many ways to improve the efficiency of the PG+CS combination, whilst PGPS can only be speeded up by improving the performance of the interpreter itself. Efficiency aside, PG's CP descriptions provide a high-level account of the fault in the student's procedure. By concentrating on those constraints which describe key aspects of the CP, the tutoring system could enter into a dialogue which is more meaningful than one based on the mal-rules in the student model. For example, the tutoring system could make statements such as: "... your algorithm works fine except when the first numerator is less than the second ...". It is not possible to derive such statements directly from the rules - effectively, all the tutoring system can say is that, in this instance, the rule **nm1** is missing.

4.9. Conclusions

PG successfully analysed the models which RPC was unable to deal with in chapter 2. It also successfully generated the I/O mappings for 20 fraction subtraction models derived from an earlier study of children's errors in that domain. Although this evaluation was successful, this would not have been the case had any of the models contained loops - this limitation is addressed in section 5.4. We discussed ways of improving the efficiency of PG's abstract interpreter, theorem prover and CP instantiator. In the next chapter, we discuss how PG could be developed further.

Chapter 5

The Way Forward

5.1. Introduction

In this thesis we have developed an algorithm which takes a pair of production rule models and derives an abstract description of the set of CPs for the model pair. This abstract CP description can then be instantiated by a suitably equipped constraint satisfaction procedure.

This chapter is fairly speculative in nature and outlines possible lines of further research. We begin by discussing the role of 'counter examples' in tutoring and point out that such problems can be generated using PG's algorithm as it stands. We have not really addressed the question of how PG might fit within the student modelling component of a tutoring system. This is remedied in section 5.3. PG's major drawback is that it cannot reason about loops in production systems; we discuss this issue in section 5.4 and outline some ways of solving this problem. One solution to the problem of CP-generation is to design a language which simplifies the process of abstract interpretation while not sacrificing too much psychological plausibility. In section 5.5, we propose Logic Programming as a potentially fruitful candidate. We conclude the chapter with a discussion of whether PG could be used to optimise and debug rulesets.

5.2. Critical Problems vs. Counter Examples

In this section, we describe another potential application of PG to tutoring systems, that of generating 'counter examples'.

The *Socratic* method is a well-established approach to the problem of helping students refine and debug their knowledge. In this paradigm, the tutor encourages students to refine their knowledge by presenting each one with instances (*counter examples*) which run counter to their conception of the domain. From analyses of actual student/teacher dialogues, Stevens and Collins (1977) have developed a set of heuristics for choosing counter examples. In general, a counter example can be viewed as an *instance* which satisfies the antecedent of the rule which the student is thought to be using, but violates the rule's consequent. Such an instance reveals that the rule is false, because it yields a false conclusion. The following example illustrates this point. The rule that all birds fly, coupled with a particular instance of a bird 'OSTRICH', leads one to conclude that ostriches fly. Now, given that we know that ostriches can't fly, it follows that the rule is in need of modification.

$$(\forall x\ \mathrm{BIRD}(x) \supset \mathrm{FLIES}(x)) \wedge \mathrm{BIRD}(\mathrm{OSTRICH}) \therefore \mathrm{FLIES}(\mathrm{OSTRICH}).$$

The difficulty of counter example generation is dependent on the domain being tutored. More specifically, for a particular abstract rule, difficulty is dependent on the computational complexity of the algorithm for computing the set of possible instantiations of that rule, and of the algorithm for evaluating the *truth value* of each rule instantiation. For example, if the student believes that all prime numbers are odd, our goal would be to find a prime number which is not odd. The set of possible instantiations is the (infinite) set of all prime numbers, and the process of evaluating each instantiation corresponds to checking whether each generated prime number is odd.

Previous uses of counter examples in tutoring systems have been restricted to areas where the domain of discourse can be represented as a relatively small set of objects. For example,

WHY represents knowledge about rainfall as a set of scripts. In such domains, counter example generation involves a manageable search through a finite set of possible instantiations. However, in many domains, such as arithmetic, the set of instantiations is *very large* and the process of evaluating the truth value of a particular instantiation can be very expensive. Consider the procedural domain of multi-column subtraction. Cast in the above framework, the set of all legal subtraction problems, paired with the answers produced by the student model, defines the set of possible rule instantiations. The student model corresponds to the mapping between the set of problems and the set of answers. The assertion that the student's algorithm is sound can be shown to be false by finding a problem instance which maps onto an incorrect answer.

In practice, because of the flexibility required of student models, they are computationally expensive to run. Therefore, it is not feasible to run the student model on each member of the set of legal problems, until a counter example is found. Furthermore, if the student's algorithm is equivalent to the tutor's ideal, but expressed differently, then the tutor will be faced with a non-terminating search for a counter example. For these reasons, one cannot always generate counter examples by enumerating concrete instances.

In order to overcome these difficulties, the tutor must be capable of reasoning about the *abstract* computational behaviour of student models. This is precisely what PG achieves when trying to find CPs. The process required to find a counter example is the same as that required to find a CP, this is because counter examples and CPs are formally equivalent. The different terminology refers to their ultimate use rather than any formal difference between them - CPs highlight the difference between two alternative student models, while counter examples highlight the difference between the student's 'faulty' model and the tutor's 'correct' one.

Although PG could be applied to the problem of generating counter examples in procedural domains, it does not address the question of what constitutes a 'good' counter example. However, one could augment PG with rules such as those of Stevens and Collins (1977). Such rules could either be used as a 'filter' on the CPs generated by PG or as a heuristic, guiding the search for CPs in much the same way as was discussed in section 4.8.3.

5.3. The Integration of PG with ITSs

The whole of this thesis has been devoted to the task of developing a method of deriving CPs from pairs of production rule models. Yet, we have not considered how such a facility would be integrated with an ITS. There are many problems with integrating PG with ITSs, for example, we have blithely assumed that the outputs of a student model can be clearly delineated in advance. However, this assumption may not be valid. In this section, we will also address the problem of generating CPs which cut down the hypothesis space, as opposed to ones which discriminate between two hypotheses only.

5.3.1. Multiple Hypothesis Testing

When a tutoring system meets a student for the first time, may not know very much about him/her. It would be unlikely to be in a position to set a CP to discriminate between two hypotheses, because it would probably not have enough information to pick two models from the many possible ones at its disposal (unless, say, the teacher had initialised the student model in some way). Its first task would be to learn enough about the student to enable it to choose the appropriate subject matter for teaching. This data collection phase of student modelling is exploratory in nature and has parallels with scientific investigation. In science, one normally tries to learn enough about the area of interest before one starts to propose theories which explain that area. Once some initial hypotheses have been generated, the scientist can then consider the design of experiments to further refine those ideas.

Our work does not address the question of how the tutoring system should reach the point where it has two candidate hypotheses. PG could be used to reduce a larger set of hypotheses to a singleton by taking pairs of models and generating tests to discriminate between each pair. However, this approach is not a practical proposition for tutoring systems because of the large number of problem examples which the student would have to solve (it would be O(M) where M is the number of different models available to the tutoring system).

An alternative approach would be to employ a 'binary search' regime. If we could generate a problem which splits the hypothesis space into two, then the number of tests would be reduced to $O(\log_2(M))$. Conceivably, there might be times when one CP would discriminate between all of the models; in such, admittedly rare, cases the number of tests would be O(1). Note that these are *theoretical* estimates; one would not expect students to be perfectly consistent in their answers, therefore, the actual number of tests required would be greater than these estimates.

The question is, given a set of candidate models, can we automatically generate a CP which either reduces that set to a singleton, or reduces the set by half? The answer to this question is that it depends on the features of the candidate set of models. The relevant features can be extracted by abstract interpretation. The first step is to derive the I/O mappings for each model by applying PG to each model individually (i.e. not to pairs of models). The I/O mappings can then be grouped in terms of their inputs - i.e. we group together those mappings whose inputs intersect (as described in section 3.8.2). For example, the following two inputs would be grouped together:

Input1: (?nx - ?ny), where ?nx > ?ny.

Input2: (?nx - ?ny), where ?nx ≥ ?ny.

However, the following two would go in a separate group:

Input3: (?nx - ?ny), where ?nx < ?ny.

Input4: (?nx - ?ny), where ?nx < ?ny, and EVEN(?nx).

It would not be sensible to try to generate a CP which discriminates between these two groups, because their inputs form two disjoint sets. Therefore, after grouping the inputs into sets where the members intersect, we would have to generate, at the very least, one CP for each group.

Let us now consider how one would generate CPs to divide up a group whose inputs intersect. If there were N candidate models in a group, then our goal would be to find an input which satisfies the following constraint expression (where O_m is the output for candidate M):

$$O_1 \neq O_2 \neq ... \neq O_n$$

This may not always be possible. For example, it might be possible to find an input for which $O_1 \neq O_2 \neq O_3$, however, for that input it might be the case that $O_4 = O_5 = ... = O_n$. Therefore, a CP could be generated which reduces the set of candidates {$Model_1$, $Model_2$, $Model_3$} to a singleton, however, more CPs would be needed to filter the remaining models {$Model_4$,...,$Model_n$}. In order to turn this into an efficient procedure, we would have to develop an algorithm, or a set of heuristics, for dividing these sets into subsets which would minimise the number of CPs required. This would be an interesting area for further research. What we are after is a method of automatically generating a highly diagnostic test from a set of candidate models.

5.3.2. When is an Output an Output?

Throughout this thesis, we have assumed that the student models explicitly flag those elements which are outputs (i.e. 'observables'). This research would have been much more time consuming otherwise. Nevertheless, we need to consider how PG would function in the 'real world', where such assumptions might be too restrictive. In principle, any WME could be viewed as a potential output of a model. PG could handle such a viewpoint, but it is likely that this would impair its CP-generation capabilities. For example, if PG were to find a CP which discriminates between two models on the basis of some intermediate WME, then the likelihood that this CP would actually discriminate in practice would depend upon the probability that the student would provide evidence of the WME in his/her protocol. Therefore, to assess the effectiveness of a CP one would have to estimate the probability that the student would include the desired output in his/her protocol.

Such an estimate could be based upon experience of the particular student, or upon experience of students in general. Alternatively, one could build in the probability estimates. For example, students are more likely to include the final answer in the protocol, because they know that this is the very least that the tutor requires. Therefore, a CP which discriminates on the basis of the final answer would have a high probability of being effective. However, there may be cases where the two models are equivalent in terms of their final answers (e.g. mental arithmetic). In such cases, PG would have to treat the intermediate steps of each model as observables. In such cases, the greater the number of differences between the intermediate steps of the two models, the greater the probability that the CP will be effective. If we assume that the likelihood of all intermediate steps appearing in the protocol is equal, then, because PG would be trying to maximise the probability of unequal outputs, it would choose a CP which produces the greatest number of unequal intermediate steps (between the two models).

Finally, we should point out that the tutor could enter into a diagnostic dialogue with the student, whereby the machine would ask the student about critical steps missing from the protocol. If the student has not revealed the critical step in his/her problem solving, then it should be possible to extract it by interrogation.

5.4. PS Loop Abstraction

In section 4.7 we introduced the problems which PG has in characterising the behaviour of iterative groups of production rules. To reiterate, there are three major problems to be solved:

(i) Identifying the group of rules which forms the loop;

(ii) Computing what changes on each loop iteration and deriving the termination condition for the loop;

(iii) Representing the abstract behaviour of the loop so that other rules in the ruleset can manipulate the abstract sequence of WM changes effected by the loop.

Each of these problems is non-trivial, although identifying a loop is easier than the other two tasks.

5.4.1. Loop Identification

In most programming languages, identifying *explicit* loops is an easy problem. High-level languages provide structured looping constructs which are used to express iteration (e.g. Pascal's WHILE loop); thus, loops are made explicit by the language. Production systems do not incorporate syntactic conventions for flagging loops, rather, iteration occurs as a side-effect of the temporal aspects of the data in WM which forces a particular control flow. If we return to the simple counting ruleset from section 4.7, we can see that there is nothing about the rules which explicitly signifies a loop.

```
(define-rule start-counting
 if (count from ?x to ?y) & ~(number ?x)
 then (number ?x) & (*output ?x))

(define-rule count
 if (count from ?x to ?y) & (number ?previous) & ~(number ?y)
 then (number (*add1 ?previous)) & (*output (*add1 ?previous)))
```

In order to glean that the rule, **count**, forms a loop, we actually have to 'mentally-model' the temporal flow of WM data through the rule. If we do this, we see that the WME **(number (*add1 ?previous))** from one cycle forms the input to the LHS pattern **(number ?previous)** on the next. This mental modelling can become quite complex, it is not sufficient to note that an RHS element of the rule, **count**, unifies with an LHS element of the same rule. To accurately identify a loop we need to perform a full abstract interpretation of the ruleset. The example below makes this clear; the addition of the rule, **halt-when-even**, means that the rule, **count**, no longer forms a loop.

```
(define-rule halt-when-even
 if (count from ?x to ?y) & (number ?previous) & (*evenp ?previous) & ~(number ?y)
 then (*halt))
```

Because **halt-when-even** is more specific than **count**, it takes control as soon as **(number ?n)** appears in WM, where **?n** is even. Thus, if **?x** is initially odd, then **count** fires onces, depositing the even number **(*add1 ?x)**, which then triggers **halt-when-even**. If **?x** is initially even, then **count** does not get a chance to fire at all, as **halt-when-even** immediately takes control once **start-counting** has deposited **(number ?x)** in WM.

Abstract interpretation enables one to identify loops by modelling the temporal flow of concrete data items in terms of abstract ones. All we need do is augment PG so that it records the fired instantiations as it goes along (PG does this already; but this information is only kept to show to the user, PG ignores it otherwise). If, down a given path, PG is about to fire an instantiation of a rule which it fired earlier in the path, then the sequence of rules between those two points constitutes a loop. Note that this instantiation must be the one chosen for firing; merely being in the Conflict Set is not enough to constitute a loop (e.g. in the above 'halt-when-even' example, **count** would fire once and would then be instantiated by its own output; but this does not represent a loop because the presence of **halt-when-even** means that the second instantiation of **count** can never fire).

Our definition of a loop is quite general. One can, however, think of other more restrictive definitions. For example, one could restrict our definition so that a repeating sequence of rules is only considered to be a loop if it processes items which it has produced on previous cycles (the **count** rule is an example of such a loop). This definition, one might argue, is 'better' because it excludes paths which are only 'apparently' loops, such as the one below. In this example, the rule **print1** outputs the items in WM of the form: **(number ?x)**. It does not process any of its own outputs (there aren't any). When run on the WM: **((number 1) (number 2) (number 3))**, it would output the numbers **1**, **2** and **3**.

```
(define-rule print1
  if (number ?x)
  then (*output ?x))
```

Whilst acknowledging that one's chosen definition of the term 'loop' is a little arbitrary, we argue that it needs to be this general if we are to capture more obscure types of loop. The above example is every bit as valid a loop as the **count** example. The rule, **print1**, is treating the items in WM as a sequence of values to be processed one at a time. This is equivalent to

generating and processing them one at a time as in **print2** below. We argue that it is unnecessarily restrictive to regard only the latter as a loop just because it processes values generated by itself on earlier cycles. **Print1** is just as much a loop, but processes the whole sequence after that sequence has been generated, rather than generating *and* processing the items one at a time.

```
(define-rule print2
 if (number ?x) & ~(= ?x 4)
 then (number (*add1 ?x)) & (*output ?x))
```

Having outlined a method of loop identification, we must now consider how to generate an abstract description of the loop from the information collected during the abstract interpretation of the path in question. A prerequisite of this goal is the ability to describe that which alters on each iteration of the loop, and under what conditions the loop terminates.

5.4.2. Looping Behaviour and Termination

Before proposing methods of reasoning about PS loops, it would be instructive to examine the work on analysing loops in other programming languages. Some of these ideas may be transferable to PSs. There is a large body of work in this area, from imperative languages such as Fortran (Waters, 1979) to declarative languages (e.g. Mycroft, 1981). With declarative languages, the analyses have been of recursion rather than iteration as such. With PSs the distinction between iteration and recursion is somewhat blurred. We can encode recursive solutions to problems by treating WM as a stack, as in the factorial example below.

```
(define-rule build-stack
 if (factorial ?n) & ~(*zerop ?n)
 then (?n) & (*delete 1) & (factorial (*sub1 ?n)))

(define-rule base-case
 if (factorial 0)
 then (*delete 1) & (result 1))

(define-rule unwind
 if (?n) & (result ?f)
 then (*delete 1) & (*delete 2) & (result (*times ?n ?f)))
```

When run on **(factorial 3)** PGPS exhibits the following behaviour, where the numbers **(1)**, **(2)** and **(3)** represent a stack (ordered by the Recency function):

```
WM1:  ((factorial 3))
  Firing:  BUILD-STACK
  WM2:  ((factorial 2) (3))
    Firing:  BUILD-STACK
    WM3:  ((factorial 1) (2) (3))
      Firing:  BUILD-STACK
      WM4:  ((factorial 0) (1) (2) (3))
        Firing:  BASE-CASE
        WM5:  ((result 1) (1) (2) (3))
      Firing:  UNWIND
      WM6:  ((result 1) (2) (3))
    Firing:  UNWIND
    WM7:  ((result 2) (3))
  Firing:  UNWIND
  WM8:  ((result 6))
```

Our counting PS, presented earlier, can be viewed as either recursive or iterative. Therefore, we will not restrict our survey to work on iteration, as we could well lose much of value by so doing. We will divide this brief review into two sections. The first covers 'loop invariants' and their use in program verification. The second covers the work of Richard Waters which analyses loops into temporally-abstract loop fragments.

Loop Invariants

The notion of a 'loop invariant' was originally introduced by Floyd (1967) and Hoare (1969). A loop invariant describes a property of the loop which holds on each and every

iteration of the loop. The principle of mathematical induction is used to prove the invariance of the loop invariant. The first step is to show that the invariant property holds after zero iterations. We then show that if the invariant property holds after iteration, n, then it holds after the iteration, n+1, and therefore that it must hold for all values of n. To further characterise the loop, one derives a description termed the 'variant' of the loop. The variant describes what changes on each iteration of the loop, and can be used to prove that the loop terminates (essentially by showing that the variant converges upon the termination condition of the loop). Finally, these intermediate conclusions are used to verify that the loop meets its specification, i.e. computes what it is supposed to.

Discovering an appropriate loop invariant is one of the most difficult aspects of this process. There can be many aspects of a loop which are invariant, but only some of these will be of use for verification. Research on automating this process has concentrated on heuristic methods which guide the search process on particular classes of loop (e.g. Dunlop and Basili, 1984).

As an example, consider the ruleset below. It computes the sum of the integers between **?x** and **?y**.

```
(define-rule start
  if (sum from ?x to ?y) & ~(current ?)
  then (current ?x) & (sum 0))

(define-rule sum
  if (sum from ?x to ?y) & (current ?i) & (sum ?n) & ~(= ?i ?y)
  then (*delete 2) & (*delete 3) & (current (*add1 ?i)) & (sum (*plus ?n ?i)))
```

A loop invariant for this ruleset is: $\mathbf{n} = \Sigma(\mathbf{i}|\mathbf{x} \leq \mathbf{i} \leq \mathbf{y})$, where the question mark prefixes have been dropped to enhance readability. The variant function is: $\mathbf{i}=\mathbf{i}'+\mathbf{1}$, where **i'** represents the value of **?i** on the previous iteration. To prove that **?i** does indeed converge on **?y** we rely on the presence of the following precondition in the loop specification: $\mathbf{x} \leq \mathbf{y}$.

Once computed, the above loop invariant would be of use to PG because it describes an important aspect of what the loop computes, i.e. the loop computes a WME of the form: **(sum ?n)**, where **?n** is the sum of the integers between **?x** and **?y**. However, as we have said, deriving loop invariants is a non-trivial problem - we would probably have to settle for a heuristic method which only handles the common classes of loop.

Loops as Stream Processors

Waters (1979) presents an interesting method of analysing loops. It is based on the observation that most loops are built from four small, stereotyped chunks (referred to as Plan Building Methods, PBMs). The analysis in terms of PBMs can be used to derive a proof of loop correctness which is much easier to produce and is more useful than a proof based on a loop invariant. PBMs allow one to break loops up into small, easily understood fragments which are combined together by the simple mechanism of composition. This methodology is based on the key insight that, logically, the temporal sequence of values processed by the loop can be treated like an aggregate data object. The temporal sequencing in a loop is merely an efficiency measure; it is more efficient to generate and process values one at a time, rather than to first create an explicit data structure which holds all of those values, and then process the items one at a time. For example, the **sumBetween** predicate sums the integers between **X** and **Y** inclusive.[1]

```
sumBetween(Y,Y,PartialSum,PartialSum+Y).
sumBetween(X,Y,PartialSum,Result) <-
  sumBetween(X+1,Y,PartialSum+X,Result).

sumList([],Result,Result).
sumList([H|T],PartialSum,Result) <-
  sumList(T,PartialSum+H,Result).
```

On each iteration, **PartialSum** is incremented by the value of **X**, and **X** is then incremented by one. The process terminates when **X** is equal to **Y**. Thus, **sumBetween** processes the temporal sequence of values which flow through **X**. The predicate, **sumList**, is an alternative definition of this relation; it sums the elements in the list which is its first argument. This predicate can be viewed as processing the *sequence* of values held in that list, the only difference is that the sequence is held in an explicit data structure, rather than an implicit,

[1] Again, this is pseudo-Prolog code. We are using '+' as if it were a function, thus, in these two examples it gets 'applied' in the functional programming sense. Expressing it this way saves one from having to use Prolog's built in 'is' predicate.

temporally-defined one. Waters would argue that, logically, **sumBetween** and **sumList** are doing the same job; furthermore, viewing **sumBetween** as though it were processing an aggregate data object simplifies the analysis of the loop. It is easy to analyse **sumList** into two orthogonal loop fragments which are composed together; similarly, by treating **X** as if it were an aggregate data object, one can express **sumBetween** as the composition of the same two loop fragments. The variable, **X**, can be viewed as a loop fragment which generates a *stream* of values which are passed to a 'summing' loop fragment. In the same way, the list passed to **sumList** can be viewed as an integer-generating stream which is composed with a summing loop fragment. This function composition can be expressed as follows, using Waters' SERIES[1] package (Steele, 1990): **(collect-sum (scan-range :from x :upto y))**. **Scan-range** generates a 'series' (analogous to a stream) of integers between **x** and **y**, and **collect-sum** reduces that series to a singleton using the operator '+'; thus, the expression computes: $\sum\{i|x \leq i \leq y\}$.

The four basic loop PBMs are:

(i) Augmentation; this extends a basic loop by processing the stream of values produced by the basic loop, but without actually altering them.

(ii) Filtering; this takes a stream of values and restricts it to some subset. For example, if we augmented **sumBetween** so that it only summed the even numbers between **X** and **Y**, then that would be expressed as a filter which only allows through those integers which are even. By composing this fragment with the others, the above SERIES expression would become: (collect-sum (collect-if #'evenp (scan-range :from x :to y))).

(iii) Basic Loop; this is the basic 'control computation' of the loop and governs when termination happens (**scan-range** in our above example).

(iv) Interleaving; this expresses the intermingling of loops so that they operate in synchrony. The loops are independent in the sense that there is no data flow between them, but they are dependent upon one another because they both terminate as soon as one of them does. As an example, consider the following example (written in an imperative style rather

[1] The SERIES package is written in Common Lisp, and allows one to do the reverse of what we are discussing in this section. Rather than analyse procedurally-defined loops into an equivalent declarative framework, SERIES enables the programmer to express his/her loops in a declarative fashion, in terms of function composition. The package takes care of the job of turning that declarative encoding into an efficient loop, by interleaving the process of generating values with that of processing them. Loop fragments as analysed into PBMs are certainly not one and the same as SERIES, however, SERIES are based on that work and provide a simple way of expressing these ideas.

than pseudo-Prolog, because it is difficult to express interleaved iteration clearly in that language).

```
  N := START;
  SUM := 0;
  DO I:= 1 TO 10
   IF N>LIMIT THEN GO DONE;
   IF PRIME(N) THEN SUM := SUM+N;
   N := N+1;
  ENDDO
DONE ...
```

This piece of code computes the sum of the prime numbers between START and LIMIT, subject to the constraint that it process no more than the first ten of them. Thus, if START were 1 and LIMIT were 5, then it would compute $\Sigma\{2,3,5\}$; but if LIMIT were 40, then it would compute $\Sigma\{2,3,5,7,11,13,17,19,23,29\}$ rather than $\Sigma\{2,3,5,7,11,13,17,19,23,29,31,37\}$. This loop can be analysed into two interleaved ones:

Loop 1

```
DO I:= 1 TO 10
ENDDO
```

Loop 2

```
  N := START;
  SUM := 0;
LOOP
  IF N>LIMIT
  THEN GO DONE;
  IF PRIME(N)
  THEN SUM := SUM+N;
   N := N+1;
  GO LOOP;
DONE ...
```

Loop 1 is a control computation which just iterates ten times. Loop 2 computes the sum of the primes between START and LIMIT. One of the payoffs of this analysis is that we can say that the loop terminates when either of the interleaved ones does; therefore, proof of the termination of either loop is a necessary and sufficient condition for proving that the interleaved loop does. Thus, the above loop terminates when **I=11 ∨ N>LIMIT**. Using SERIES, the interleaved loop is easily expressed as the composition of several fragments; note that we use the range 0 to 9, rather than 1 to 10, because SERIES functions assume a zero origin.

```
(collect-sum (subseries (choose-if #'prime (scan-range :from start :to limit)) 0 9))
```

In summary, to verify the correctness of a loop, it is analysed into loop fragments and an invariant is found for each fragment. Because each fragment is so simple, discovering these invariants is easier than the usual task of finding a single invariant for the loop as a whole. Waters does not claim that this method can find the invariant for any loop, but rather that it greatly simplifies the task for most loops. Furthermore, because the loop is fragmented, if the loop fails to meet its specification, it is that much easier to localise the source of the problem.

We shall now develop some examples of PS loops and see whether any of the above ideas are of use in their analysis.

5.4.3. A Simple PS Loop

Let us begin by returning to the simple counting ruleset from section 4.7. In section, 5.4.1 we noted that it would not be difficult for PG to identify that the rule, **count**, forms a loop. If we started with the Input Specification, **(count from ?a to ?b)**, then during abstract interpretation, PG would fire **count** once and would then find that it was instantiated and ready to fire on the next cycle. This latter instantiation would contain the following Negation Nullifier: **¬EQ(?b.0,add1(?a.0))**, this is because the negated pattern, **~(number ?y)**, has been unified with the WME, **(number (*add1 ?previous.2))**, deposited after the previous firing of **count1**, where **?previous.2/?a.0** is in the environment. The rule can only fire again if the Negation Nullifier is satisfied.

```
(define-rule start-counting
  if (count from ?x to ?y) & ~(number ?x)
  then (number ?x) & (*output ?x))

(define-rule count
  if (count from ?x to ?y) & (number ?previous) & ~(number ?y)
  then (number (*add1 ?previous)) & (*output (*add1 ?previous)))
```

If we freeze the abstract interpretation at this point, the binding environment for the variables in the rule provides some clues as to what this loop is doing. After the first firing of **count** the environment is as follows:

(?previous.2/?a.0 ?y.2/?b.0 ?x.2/?a.0 ?y.1/?b.0 ?x.1/?a.0)

The next, about to be fired, instantiation of **count** has the following environment:

(?previous.3/(*add1 ?a.0) ?y.3/?b.0 ?x.3/?a.0
?previous.2/?a.0 ?y.2/?b.0 ?x.2/?a.0 ?y.1/?b.0 ?x.1/?a.0)

If we examine the data flow through these variables, it is clear that the variable, **?previous.n**, becomes wrapped in the expression **(*add1 ?previous.n-1)**. It is quite simple to keep a record of the data flow between variables by storing an auxiliary environment which holds the unification which would have been performed had we not used the 'ultimate associate' (section 3.5.3). For example, when **(number ?previous.2)** was being unified with **(number ?x.1)**, the algorithm computes the ultimate associate of **?x.1** before creating a binding. The ultimate associate of **?x.1** is **?a.0**, thus **?previous.2** gets bound to **?a.0**. However, we could store the binding between **?previous.2** and **?x.1** in an auxiliary environment, yielding the following environment for the second instantiation of **count**:

(?previous.3/(*add1 ?previous.2) ?y.3/?b.0 ?x.3/?a.0
?previous.2/?x.1 ?y.2/?b.0 ?x.2/?a.0 ?y.1/?b.0 ?x.1/?a.0)

This makes the data flow through the variable, **?previous**, more explicit. This data flow is expressed diagramatically in figure 5-1. The variables **?x** and **?y** are initialised to **?a** and **?b** respectively, and keep those two values throughout the loop's duration. The variable, **?previous**, is initialised to **?a** and an output and a WME are generated, provided that

¬EQ(?previous,?y). The expression, (*add1 ?previous), forms the subsequent input to ?previous.

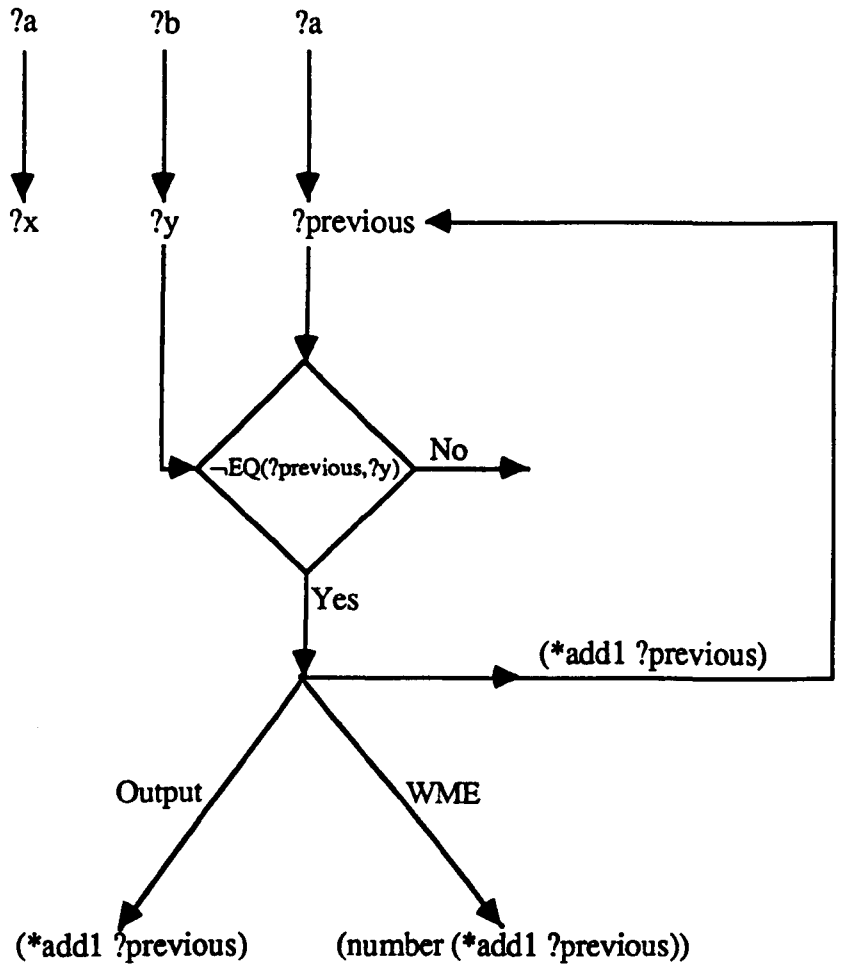


Figure 5-1 – Data/control flow for the COUNT rule

It would not be difficult on the basis of this data/control flow analysis to derive the following SERIES description of the loop:

```
(map-fn 'output
  #'(lambda (n) `(*output ,(1+ n)))
  (scan-range :from x :below y))

(map-fn 'wme
  #'(lambda (n) `(number ,(1+ n)))
  (scan-range :from x :below y))
```

In each case, the function **map-fn** maps the lambda expression over the elements produced by **scan-range** and produces a series of values, one of type 'output' and the other of type 'wme'. The **:below** keyword signifies that the series is bounded by y but does not include that value. The abstract series of outputs can be stored along with the other outputs collected on this path, yielding an abstract description of the temporal sequence of outputs produced by the ruleset. Likewise, the abstract series of WMEs could be added to WM, but this leaves us with the problem of specifying how PG would manipulate such a datum and what such a datum would look like. A syntactic variant of the above SERIES description would do; it would need to contain a description of the form of the WMEs deposited, a description of how the WMEs change from cycle to cycle and a description of the series' bounds. For example, a description of the **count** loop, based on the Input Specification **(count from ?a to ?b)**, could look as follows:

```
(#series :wme (number (*add1 ?n))
   :with ?n :from ?a.0 :below ?b.0)
```

We need to outline how this datum would be handled by PG - it is very different from the sort of abstract WMEs PG processes at the moment. Consider the effect of adding the following rule to the counting ruleset:

```
(define-rule even
 if (number ?z) & (*evenp ?z)
 then  (*output (?z is even)) & (*halt))
```

Due to Specificity, **count** always wins during conflict resolution. However, once **count** has reached ?y, the rule, **even**, can fire The instantiation which fires will be the one which matched the most recent even number in WM. Thus, the result of unifying the pattern (**number** ?z) with the above series should be a state which expresses this fact:

```
?z/(collect-first (choose-if #'*evenp (reverse-series (scan-range :from ?a.0 :upto ?b.0))))
```

The above SERIES expression denotes the fact that the variable, ?z, gets bound to the last even element of the series. In general, because of Recency, series elements will be processed in reverse order by other rules.

In this example, it was not difficult to identify the termination condition of the loop - it terminates when one of the LHS constraints of the rule is violated. It seems likely that PG's abstract interpretation mechanisms could identify the loop termination condition with very little modification. A PS loop terminates when one of its LHS constraints is violated; PG can already synthesise an expression which describes this condition - it does so when building an 'exclusion clause' during conflict resolution (section 3.7.2). The exclusion clause specifies the conditions under which a given rule could override another which would otherwise have been selected during conflict resolution. The loop termination problem is a subset of the exclusion clause generation one. Rather than build a clause which expresses the conditions under which a rule would be prevented by another from firing, we build a clause which expresses the conditions under which a rule could not fire, regardless of the other rules in the ruleset. To illustrate, two examples of termination condition generation are given below.

LHS: (check if double) & (?x ?y) & (= ?y (*times ?x 2)) & ~(already-checked ?x ?y)

WM: ((check if double) (?a ?b) (already-checked ?a ?c))

Termination Condition: ¬EQ(?x,?a) ∨ ¬EQ(?y,?b) ∨ ¬EQ(?b,(*times ?a 2)) ∨ ¬EQ(?b,?c)

LHS: (?x isa ?y) & (?y isa ?z) & (*category-p ?y) & (*category-p ?z)

WM: ((?a isa ?b) & (primate isa ?c))

Termination Condition: ¬EQ(?x,?a) ∨ ¬EQ(?b,primate) ∨ ¬EQ(?z,?c) ∨ ¬CATEGORY(?z)

Note that, because the termination condition is developed during abstract interpretation, it is expressed relative to the current state of WM. Thus, it might be quite different down another abstract interpretation path of the ruleset. This allows PG to develop quite specific termination conditions, such as the second one above. Because of the presence of **(primate isa ?c)** in WM, we can generate a termination disjunct which says that the rule will not fire if **¬EQ(?b,primate)**; similarly, we do not have to include the **CATEGORY** constraint on **?y**, because if **EQ(?b,primate)** is true, then **¬CATEGORY(?y)** can be determined by evaluation, because ?y's binding is ground.

Identifying a loop termination condition seems to be a subset of the problem of generating exclusion clauses. Of course, identifying termination is quite different from the problem of proving it. That program termination is a semi-decidable issue is a well-known result of computation theory. The question is, does PG have to *prove* that a loop terminates, or is it sufficient to identify the conditions under which it would. We would argue that a proof is not always a necessary prerequisite for generating CPs; but it can yield useful extra information about the path being explored by PG. The counting ruleset terminates when it reaches **?y**, but if **?x** is initially greater than **?y**, then the **count** loop would be non-terminating. Our current analysis does not make this fact clear. The advantage of developing a proof of termination is that it would have to be based on the assertion, **x≤y**. Thus, PG would have to derive this assertion as a precondition on the loop; this precondition is crucial to the generation of a valid I/O mapping. In general, we would expect important preconditions, such as **x≤y**, to be part of the Input Specification which the ITS uses to describe the problem space. Therefore, failure to prove loop termination would not be as damaging as it seems.

As an example of how PG might prove loop termination, consider the following ruleset:

```
(define-rule subtract
  if (number ?x) & (> ?x -10)
  then (dividend (*subtract ?x 10)))

(define-rule divide
  if (dividend ?y)
  then (number (*divide ?y 2)))
```

This could be analysed into the following expression:

```
(#series :wmes ((number (*divide (*subtract ?x 10) 2)) (dividend (*subtract ?x 10)))
   :with ?x :from ?a.0 :by (*divide (*subtract ?x 10) 2) :above -10)
```

From this description, we can see that **x** is changing according to the function: **x=(x-10)/2**. Given that subtraction and division are both monotonic functions, we can determine the fixed point for this composed function by solving for **x**:

$$x = (x-10)/2$$

$$\therefore\ 2x = x-10$$

$$\therefore\ x=-10$$

The expression, **x=-10**, is the fixed point of the function, so when **x** is **-10**, the function always outputs **-10**, i.e. it is non-terminating (**x** never changes). However, because of the precondition on the loop (**x > -10**), the loop is never entered. But when does the function's output decrease? Well, by incrementing **x** by a small amount we can determine in which direction the function's output goes. If we make **x=-9** initially, then after one iteration its value becomes **(-9-10)/2**, i.e. **-9.5**, so if **x** is initially greater than the fixed point, then it decreases on each iteration. Thus, the above loop will converge on the fixed point.

This only scratches the surface of the problem of verifying loop termination for PSs; however, on the basis of this analysis it does not seem to be an insoluble problem.

5.4.4. PS Loop Composition

We shall now consider a more complex example, in which the WMEs generated by one loop are processed by another. This ruleset consists of the two rules, **start-counting** and **count**, plus the following two extra rules:

```
(define-rule start-summing
 if (sum even numbers) & ~(partial-sum ?)
 then (partial-sum 0))

(define-rule sum-evens
 if (partial-sum ?sum-so-far) & (number ?n) & ~(used ?n) & (*evenp ?n)
 then (partial-sum (*plus ?n ?sum-so-far))
   & (used ?n)
   & (*output (*plus ?n ?sum-so-far)))
```

Once **count** has deposited all of the integers between **?x** and **?y** in WM, **start-summing** takes over and deposits **(partial-sum 0)**. This triggers **sum-evens** which processes the elements of the form **(number ?n)** in order of recency. Each time around the loop, if **?n** is an even number, then it increments the **?sum-so-far** by **?n** and outputs that partial sum. The loop terminates when all of the elements of the form **(number ?n)** have been processed.

This loop is fairly similar to the **count** one, but processes a sequence of WMEs rather than temporally generating its own sequence. Because of the Recency rule, it processes the data from the **count** series in reverse order, thus, the **sum-evens** series would look something like the following:

```
(#series :wmes ((partial-sum (*plus ?n ?sum-so-far)) (used ?n))
   :with ?n :choose-if *evenp :from ?b.0 :downto ?a.0
   :with ?sum-so-far :from 0 :by ?n)
```

If we added another rule which picked up the final sum, output it and halted, then the expression describing the final output would look as follows:

```
(collect-sum (choose-if #'*evenp (reverse-series (scan-range :from ?a.0 :upto ?b.0))))
```

The **scan-range** form is derived from the **count** series and must be reversed (because of Recency). This is composed with an operation which selects the even elements of the series and sums them.

5.4.5. Conclusions

In this section, we have touched on some of the problems of reasoning about PS loops in abstract domains. Given our definition of PS loops, loop identification and expressing loop termination seem to be soluble problems, although, in order to derive an accurate description of some loops, it will be necessary to prove their termination. We will also need to solve the problem of representing cases where some of the elements produced by a loop get deleted later on by other rules.

Augmenting PG to reason about loops would be an undertaking of a magnitude equal to or greater than the work described in this thesis. Nevertheless, we have not discovered anything peculiar to PSs which would make this problem insoluble. Our cursory solutions to some of these problems are based on the Waters' approach, because this seems to fit well with the current form of PG. To mesh with PG, an analysis of loops should make the temporal sequence of values, generated by loops, explicit. This can be done by creating an aggregate data object which encapsulates a description of the sequence of elements generated by the loop.

It is worth pausing for a moment to consider what our goals for loop analysis are. We want to be able to analyse student models which contain loops. However, such models are unlikely to be very complex. Our **sum-evens** example from the previous section has very little psychological plausibility. The psychological constraints on Working Memory mean that it would be infeasible to sum even numbers by first generating a sequence of integers, and then adding those members which are even. One would tend to generate, filter and add each number in turn, thus, complex instances of loop composition would be unlikely to occur in

real student models. Of course, if student's were making use of some kind of external memory (e.g. pencil and paper), then this limitation would not apply.

5.5. CP-facilitating Language Design

Most of the effort of this research was devoted to the problem of developing a program which can derive abstract I/0 mappings for models expressed as production systems. We chose PSs as our representation because of the many claims made for their value in modelling procedural skills. However, PSs were not designed with abstract interpretation in mind which may be one reason why PG's algorithm is as complex as it is. One avenue which suggests itself is to design a language which is both good for student modelling *and* easier to reason about in the abstract. If well designed, this language might simplify some of the problems which complicate PG's design (e.g. the use of exclusion clauses to handle conflict resolution, or the difficulty of representing the deletion of elements generated by loops).

It is difficult to design a language for student modelling without some idea of the domains we want to represent. In keeping with the rest of this chapter, we will just outline some ideas on this topic, leaving the hard part to those who choose to take it further.

Although it not feasible to fully specify a new language here, we can at least discuss the features we would want this language to incorporate. If we restrict ourselves to procedural models and take PSs as an ideal to be aspired to, then we can propose the following as important features for our new language:

(i) it should make use of pattern-directed invocation;

(ii) one should be able to model at various levels of detail (i.e there should be a choice of modelling-grain-size);

(iii) knowledge should be divided into discrete chunks, so that the tutoring system can substitute buggy ones for good ones;

(iv) deletion of parts of the model should not imply that the model ceases to function;

(v) it should make the load on WM explicit so that models are restricted to psychologically plausible ones.

In addition to the above modelling considerations, we also want the language to be easier to analyse than PSs. The family of Logic Programming languages would seem to satisfy this latter criterion. There is already a large body of work on the abstract interpretation of such languages (e.g. Mellish (1986); Jones. & Sondergaard (1987); Höök (1988)). Thus, we might be able to borrow from these approaches. Surprisingly, the use of some members of this family of languages for student modelling, has much to recommend it from a psychological perspective. In particular, we would argue that, under certain conditions, such a language could satisfy criteria (i) to (iv). This approach has already been adopted as a representation of student models of mental arithmetic (Hennessy et al, 1989). Heulog, the Logic Programming language developed in that study, is a heuristic search theorem prover, which works on Horn Clauses. It is essentially Prolog with a heuristic rather than depth-first control structure. Because the search is intelligent, rather than depth-first, models do not have to have their rules ordered (unlike Prolog). The rules have a much more declarative flavour, control information being abstracted-out in the form of heuristics. The heuristics are used to guide the search for a set of clauses which accounts for the student's behaviour.

The major drawback with this language is that it is not obvious how to represent the notion of Working Memory load because there is no such explicit structure. One solution to this problem might be to measure the 'processing depth' of the model (i.e. the number of clauses invoked down the current path). However, this is unsatisfactory - under this interpretation, counting to 100 would be memory intensive. The critical feature is not the depth of a solution, but the amount of information which must be saved as the solution unfolds. The following pseudo-Prolog definition of counting involves no memory overhead over and above that required to store the current number and the stopping point:

```
count(X,X).
count(X,Y) <-
  X≠Y, count(X+1,Y).
```

Neither does this definition of factorial require caching more than the current number and the cumulative product:

```
factorial(N,F) <- fact(N,1).
fact(0,F).
fact(N,F) <-
  N≠0, fact(N-1,F*N).
```

In contrast, this non-tail-recursive method requires a stack roughly equal in length to the initial size of N.

```
factorial(0,1).
factorial(N,F) <-
  N≠0, factorial(N-1,F1), F is F1*N.
```

In summary, on the basis of the work on the abstract interpretation of Logic Programming languages and the use of Heulog to model mental arithmetic, this would seem to be a very promising route to follow.

5.6. Further Applications of PS Abstract Interpretation

In this section, we will discuss potential applications of PG to two quite different problems: PS optimisation and debugging. Each of these problems has its own features which will entail major modifications to PG, however, they seem to be fruitful areas for research.

5.6.1. Using Abstract Interpretation to Optimise Rulesets

In section 5.4, we considered how to augment PG to handle loops. If we assume, as seems likely, that it is possible to achieve this goal, then one could use this mechanism to improve the efficiency of production rule interpreters. If a loop can be transformed into a SERIES expression, then that part of the ruleset would run much faster. The Rete match algorithm is

extremely fast - in the decade since its publication, researchers have struggled to make small improvements to its efficiency. The Rete match algorithm achieves its speed by transforming a PS into a representation which allows fast execution on serial digital computers. This makes Rete code faster than a naive implementation of the rules in a conventional language, such as 'C', would be. If each rule were encoded in C using the 'If Then Else' construct, then, for most rulesets, Rete win hands down because of its ability to perform repeated condition tests only once. Thus, a naive program transformation mechanism which translates PSs into C, would not be able to compete with Rete.

The ability to analyse loops will provide us with the opportunity of speeding Rete further. For example, the factorial example, in section 5.4.2, could be transformed into the following SERIES expression:

```
(collect-product[1] (scan-range :from 1 :upto n))
```

In Common Lisp, this expression will run in time comparable to that for expressions written using Common Lisp's low-level iteration constructs (because SERIES compile into efficient forms of these very same constructs). Thus, the three factorial rules could compile into the following single rule:

```
(define-rule factorial
  if (factorial ?n)
  then (result (*eval (collect-product (scan-range :from 1 :upto ?n)))))
```

The larger the loop the bigger the payoff. Note that one would not need to prove termination for loops which are to be transformed - if a group of rules is non-terminating, then so will its transformation be, after all, we are not proposing that the abstract interpreter remove bugs also (this question is addressed in section 5.6.2).

[1] The function, collect-product, is not defined in SERIES but could be defined as follows:

```
(defun collect-product (integer-series)
   (declare (optimizable-series-function 1))
   (collect-fn 'integer #'(lambda () 1) #'* integer-series))
```

On the face of it, we could take this idea further and collapse paths which do not incorporate loops. PG's I/O mappings represent a functional description of the behaviour on each path. Therefore, one might suppose that those descriptions could be treated as executable Lisp; they can, but there are two reasons why this would not be as useful as it sounds. For one, it is likely that there would be considerable redundancy in the constraint expressions which apply across the I/O mappings of the ruleset. For example, consider the I/O mappings which PG found for the model CORRECT-H1 (Appendix IVe):

```
Input: (((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D))
Output1: ((*CANCEL-NM (*SUBTRACT ?NX ?NY) ?D)
          / (*CANCEL-DN (*SUBTRACT ?NX ?NY) ?D))
Constraints: ¬EQ(CANCEL-NM(SUB(?nx,?ny),?d),0) ∧ ¬EQ(SUB(?nx,?ny),0)
∧ NEEDS-CANCELLING-P(SUB(?nx,?ny),?d) ∧ LT(SUB(?nx,?ny),?d) ∧ LT(?ny,?nx)
∧ LT(?nx,?d) ∧ LT(?ny,?d)) ∧ ¬EQ(?nx,?ny)

Output2: ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)
Constraints: ¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))
∧ (EQ(SUB(?nx,?ny),0) ∨ ¬NEEDS-CANCELLING-P(SUB(?nx,?ny),?d) ∨ ¬LT(SUB(?nx,?ny),?d))
∧ ¬EQ(?nx,?ny)
```

If we transformed these constraints into the condition parts of an 'If Then ElseIf' statement, then it is clear that a lot of redundant tests would be performed in the ElseIf condition. nevertheless, the redundancy is open to optimisation; it seems fairly straightforward to remove the redundancy to yield the code below. The common constraints have been removed from each constraint expression and placed in the outer IF_1 statement. Further optimisations are possible, for example, the constraints in the IF_3 statement are merely the negation of those in the IF_2 one and so can be optimised out.

```
IF1     ¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d)) ∧ ¬EQ(?nx,?ny)
THEN1   IF2   ¬EQ(CANCEL-NM(SUB(?nx,?ny),?d),0)
              ∧ NEEDS-CANCELLING-P(SUB(?nx,?ny),?d) ∧ LT(SUB(?nx,?ny),?d)
        THEN2 ((*CANCEL-NM (*SUBTRACT ?NX ?NY) ?D)
                / (*CANCEL-DN (*SUBTRACT ?NX ?NY) ?D))
        ELSE2 IF3     ¬NEEDS-CANCELLING-P(SUB(?nx,?ny),?d)
                      ∨ ¬LT(SUB(?nx,?ny),?d))
              THEN3 ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)
```

The major problem with deriving I/O mappings is the large size of rulesets used in forward-chaining Expert Systems. Rulesets of the order of 10^3 are not uncommon - one expects there to many alternative paths through them. It seems probable that it would be infeasible to search all of the abstract paths through a ruleset of that size. Furthermore, PG's need for an Input Specification might be seen by knowledge engineers as unacceptable. Therefore, much more work needs to be done before we can say whether PG's abstract interpretation mechanisms could be used to optimise rulesets. One solution might be to have the system keep audit trails of paths followed when the ruleset is run on concrete data. This empirical data could be used by the machine to select commonly-followed paths for optimisation, and would reduce the analysis required while also making the most of its results.

In effect, we are proposing a rule 'chunking' method. This approach could be viewed as a form of Machine Learning and as such is related to the notion of 'rule composition in ACT* (Anderson, 1983), 'chunking' in SOAR (Laird et al, 1986), 'macro-operators' (Korf, 1985) and 'explanation-based learning' (Mitchell et al, 1986). ACT*'s rule composition method is part of a wider psychological theory and relies on the presence of a goal structure. SOAR is also a psychological theory of learning whose roots lie in the work of Newell and Simon (1972). Korf (1985) describes a system which learns to solve problems such as the Rubik's cube by building macro-operators. Our collapsed sequences of rules can also be viewed as macro-operators. Explanation-based learning (Mitchell et al, 1986) mechanisms improve the efficiency of knowledge structures by collapsing frequently made inferences into macro-inferences.

5.6.2. Abstract Interpretation for Debuggers

We mentioned in section 3.3 that Symbolic Evaluation has been used as a basis for debugging users' code (e.g. Laubsch and Eisenstadt (1981); Wertz (1982)). Given that we have developed what is effectively a program analysis tool for PSs, one wonders whether it could be applied to the problem of debugging 'faulty' rulesets. A PS debugger could take many forms, but we discuss just three potential applications of PG to this problem: the detection of unreachable rules, spotting non-terminating paths and comparing student programs to a stored I/O specification.

Unreachable rules can be a source of bugs in PSs (cf. Nguyen *et al.*, 1985). These could be found by running PG. However, this would be problematic for large rulesets. One possible compromise would be to use a less thorough form of analysis. RPC (chapter 2) could be used to find a rule which cannot be reached by virtue of the fact that no RHS patterns unify with its LHS. This would miss rules which cannot be reached by virtue of conflict resolution. For example, one might have written a rule which is more general than another but will never fire, because, for the set of inputs which the ruleset handles, the more specific rule is always instantiated. Previous work on the analysis of rule-based systems has ignored the issue of conflict resolution (e.g. Ginsberg, 1988; Rousset, 1988; Meseguer, 1990), therefore PG can reason about an aspect of rule bases which is currently ignored (this application of PG is developed further in Evertsz, 1991).

It would be unfortunate if, because of the computational complexity involved, a PS debugger could not reason about conflict resolution. Conflict resolution (especially Recency) is one the most difficult aspects for humans to deal with when debugging faulty rulesets. The computational complexity question could be addressed by having the user highlight the rules which they believe to be problematic.

If PG were extended so that it could effectively handle loops, then it could be applied to the problem of spotting non-terminating paths. Another idea is to use PG to analyse student programs into I/O mappings which could then be compared with a stored correct set of I/O mappings. The differences could then be used to help the student debug his/her program.

These ideas hint at some of the ways that abstract interpretation could help programmers debug their rulesets. More work needs to be done before we can say whether this line of research would bear fruit.

5.7. *HALT

Much more could be said on the issues raised in this chapter, but it is time to call a halt. In this chapter we saw that the problem of generating a CP is in fact equivalent to that of generating a counter example which highlights the failings of a student model. We also touched on the problems to be faced in integrating PG with an ITS.

Loops cause special problems during abstract interpretation; although we have not implemented a solution to these problems, there are promising avenues open to further investigation.

One possible solution to the problems of reasoning about the I/O behaviour of student models is to design a language which facilitates this process.

Of course production systems are also used in areas other than student modelling (e.g. forward-chaining inference engines). The techniques employed by PG could also be used to reason about the computational behaviour of such systems, leading to better compilation and debugging facilities.

Chapter 6

Conclusions

We started this research with the goal of developing a general-purpose method of discriminating between the hypotheses of a student modelling system and chose production systems as a formalism in which to represent the models. This overall goal was simplified to the two hypothesis case because that goal enables us to solve the multiple hypothesis case also.

Scientific theories are easier to test if they make the relationships between *observables* explicit. We used this insight to develop a method of transforming production rule models into a representation which makes explicit the relationship between the input and output variables of the models. These I/O mappings can then be used to describe the set of problem examples which discriminate between the two models.

The implementation of the algorithm in Chapter 3 successfully tackled 20 empirically-based models from the domain of fraction subtraction. The algorithm for the abstract interpretation of production systems represents the main contribution of this thesis.

Though successful with the fraction models, the algorithm would not be able to cope with models which contain loops. This difficulty does not seem insurmountable, but it needs to be tackled if PG is to handle models which incorporate loops. Thus, reasoning about PS loops is an important area for further work.

Our implementation is very inefficient, thus it would be worthwhile devoting some effort to the problem of improving its efficiency. Due to the 'side-effecting' of Working Memory and

the need for conflict resolution, production systems are not particularly easy to interpret in the abstract. We suggested developing a language which is both good for student modelling and easy to reason about in the abstract. This is a very promising avenue for future research.

Our work can be viewed as part of a trend away from mere syntactic analyses of student errors. Abstract Interpretation has the potential to provide the tutoring system with access to the *semantics* of the student's procedure. For convenience, in this thesis we have been assuming that there is a single *correct* model for the domain being tutored. However, in general there are many ways of solving problems in a domain - the tutor's algorithm is just one point in such a space. Potentially, the analysis yielded by Abstract Interpretation enables the tutor to spot when the *semantics* of the student's procedure, and its own, are equivalent. Therefore, it can avoid the 'fitting a square peg into a round hole' tendency, in which the tutor forces the student to adopt its preferred algorithm.

References

Anderson, J.R. (1976) Language memory and thought Lawrence Erlbaum Associates.

Anderson, J.R. (1983) The Architecture of Cognition Harvard University Press, Cambridge, Mass.

Ashlock (1976), R.B. Error Patterns in Computation - a semi-programmed approach Charles Merrill Publishing Co., Columbus, Ohio.

Bennett, M. (1976) SUBSTITUTOR: A Teaching Program (unpublished project report), Dept of Artificial Intelligence, University of Edinburgh.

Brown, J.S. & Burton, R.R. (1978) Diagnostic models for procedural bugs in mathematical skills Cognitive Science, 2, pp155-192.

Brown, J.S. & VanLehn, K. (1980) Repair theory: a generative theory of bugs in procedural skills Cognitive Science, 4, pp379-426.

Burton, R.R. (1982) Diagnosing bugs in a simple procedural skill In D.H. Sleeman and J.S. Brown (eds), Intelligent Tutoring Systems, London: Academic Press.

Burton, R.R. & Brown, J.S. (1982) An investigation of computer coaching for informal learning activities in D.H. Sleeman and J.S. Brown (eds), "Intelligent Tutoring Systems", Academic Press, London, pp79-98.

Caferra, R., Eder,E., Fronhoefer, B. & Bibel, W. (1984) Extension of Prolog Through Matrix Reduction Proceedings of ECAI-84, Pisa, Italy, pp101-104.

Carbonell, J.R. (1970) Mixed Initiative Man-Computer Instructional Dialogues (report 1971), Bolt, Beranek and Newman, Boston, Mass.

Clocksin, W.F. & Mellish, C.S. (1981) Programming in Prolog Springer-Verlag.

Copi, I.M. (1953) Introduction to Logic The MacMillan Company, pp417-425.

Cousot, P. & Cousot, R. (1977) Abstract interpretation: a unified framework for static analysis of programs by construction or approximation of fixpoints Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages, Los Angeles, California, pp238-252.

Davis, E. (1987) Constraint Propagation with Interval Labels Artificial Intelligence, 32, pp281-331.

DeKleer, J. (1986) An Assumption-based Truth Maintenance System Artificial Intelligence, 28, pp127-162.

DeKleer, J. & Williams, B.C. (1987) Diagnosing multiple faults Artificial Intelligence, 32, pp97-130.

Downes, L.W. & Paling, D. (1958) The Teaching of Arithmetic in Primary Schools Oxford University Press.

Dunlop, D.D. & Basili, V.R. (1984) A Heuristic for Deriving Loop Functions IEEE Transactions on Software Engineering, SE-10.

Evertsz, R. (1982) A Production System Account of Children's Errors in Fraction Subtraction Technical Report #28, CAL Research Group, The Open University, U.K.

Evertsz, R. (1984) POPSI: A Packet-Oriented Production System Interpreter HCRL Technical Report #11, The Open University, U.K.

Evertsz, R. (1989a) Refining the Student's Procedural Knowledge Through Abstract Interpretations Proceedings of the 4th International Conference on AI and Education, IOS, pp101-106.

Evertsz, R. (1989b) The Generation of 'Critical Problems' by Abstract Interpretations of Student Models Proceedings of the 11th International Joint Conference on Artificial Intelligence, Detroit, pp483-488.

Evertsz, R. & Elsom-Cook, M. (1990) Generating Critical Problems in Student Modelling in M. Elsom-Cook (ed), "Guided Discovery Tutoring - A Framework for ICAI Research", Paul Chapman Publishing, pp216-234.

Evertsz, R. (1991) The Automated Analysis of Rule-based Systems, Based on Their Procedural Semantics. Proceedings of the 12th International Joint Conference on Artificial Intelligence, Sydney.

Floyd, R.W. (1967) Assigning meaning to programs Proceedings of the Symposium on Applied Mathematics, 19, pp19-32.

Forgy, C.L. (1982) Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem Artificial Intelligence, 19, 1, pp17-38.

Forgy, C.L. & McDermott, J. (1977) OPS, a domain-independent production system language Proceedings of the 5th International Joint Conference on Artificial Intelligence, Cambridge, Mass, pp933-939.

Goldstein, I.P. (1982) The genetic graph: a representation for the evolution of procedural knowledge in D.H Sleeman and J.S. Brown (eds), "Intelligent Tutoring Systems", Academic Press, London, pp51-77.

Hennessy, S., O'Shea, T., Evertsz, R. & Floyd, A. (1989) An Intelligent Tutoring System Approach to Teaching Primary Mathematics Education Studies in Mathematics, 20, Kluwer Academic Publishers, pp273-292.

Ginsberg, A. (1988) Knowledge Base Reduction: A New Approach to Checking Knowledge Bases for Inconsistency and Redundancy. AAAI88, St Paul.

Hoare, C.A.R. (1969) An axiomatic basis for computer programming Communications of the ACM, 12, pp576-583.

Höök, K. (1988) Using abstract interpretation to help novice Prolog programmers Draft document.

Johnson, W.L. & Soloway, E. (1985) PROUST - An automatic debugger for Pascal programs BYTE, April issue, pp179-190.

Jones, N.D. & Sondergaard, H. (1987) A semantics based framework for the abstract interpretation of Prolog in S. Abramsky and C. Hankin (eds) "Abstract Interpretation of Declarative Languages", Ellis Horwood, pp123-142.

Korf, R.E. (1985) Learning to Solve Problems by Searching for Macro-Operators Research Notes in Artificial Intelligence, 5, Pitman Publishing Ltd.

Kowalski, R. (1975) A proof procedure using connection graphs Journal of the ACM 22, 4, (October), pp572-595.

Laird, J.E., Rosenbloom, P.S. & Newell, A. (1986) Chunking in Soar: The Anatomy of a General Learning Mechanism Machine Learning, 1, pp11-46.

Langley, P. and Ohlsson, S. (1984). Automated Cognitive Modelling Proceedings of the AAAI National Conference on Artificial Intelligence, pp193-197.

Langley, P., Ohlsson, S. & Sage, S. (1984) A Machine Learning Approach to Student Modelling CMU Technical Report, CMU-RI-TR-84-7, Carnegie Mellon University.

Laubsch, J.H. & Eisenstadt, M. (1981) Domain specific debugging aids for novice programmers Proceedings of the 7th International Joint Conference on Artificial Intelligence, Vancouver.

Matz, MN. (1982) Towards a process model for high school algebra errors in Sleeman and Brown (eds), "Intelligent Tutoring Systems", Academic Press, London, pp25-50.

Mellish, C.S. (1986) Abstract Interpretation of Prolog Programs Proceedings of the Third International Conference on Logic Programming. London, Springer-Verlag, pp463-474.

Meseguer, P. (1990) A New Method to Checking Rule Bases for Inconsistency: A Petri Net Approach. ECAI90, Stockholm, pp437-442.

Milner, R. (1980), A Calculus of Communicating Systems. Lecture Notes in Computer Science 92, Springer-Verlag.

Mitchell, T.M. (1978) Generalization as search Artificial Intelligence, 18, 2, pp203-226.

Mitchell, T.M., Keller, R.M. & Kedar-Cabelli, S.T. (1986) Explanation-Based Generalization: A Unifying View Machine Learning, 1, pp47-80.

Motta, E., Eisenstadt, M., West, M., Pitman, K. & Evertsz, R. (1986) KEATS: The Knowledge Engineer's Assistant HCRL Technical Report #20, The Open University, U.K.

Murray, N.V. (1982) Completely non-clausal theorem proving Artificial Intelligence 18, 1 (January), pp67-85.

Mycroft, A. (1981) Abstract Interpretation and Optimising Transformations for Applicative Programs PhD Thesis, University of Edinburgh.

Neves, D.M. (1981) Learning procedures from examples Doctoral Dissertation, Department of Psychology, Carnegie Mellon University.

Newell, A. (1973) Production systems: models of control structures in W.G. Chase (ed) "Visual Information Processing" Academic Press, New York.

Newell, A. (1980) Reasoning, problem solving and decision processes: the problem space hypothesis in R. Nickerson (ed) "Attention and Performance", Laurence Erlbaum Associates, Hillsdale, NJ.

Newell, A. & Simon, H. (1972) Human Problem Solving Prentice-Hall, Englewood Cliffs, NJ.

Nguyen, T.A., Perkins, W.A., Laffey, T.J. & Pecora D. (1985) Checking an expert system knowledge base for consistency and completeness. Proceedings of 9th International Joint Conference on Artificial Intelligence, pp375-378.

Nilsson, N.J. (1980) Principles of Artificial Intelligence Springer-Verlag.

Payne, S.J. & Squibb, H.R. (1988) Understanding algebra errors: the psychological status of mal-rules CERCLE Technical Report #43, University of Lancaster, U.K.

Priest, T. & Young, R.M. (1988) Methods for evaluating micro-theory systems in J. Self (ed) "Artificial Intelligence and Human Learning - Intelligent Computer-aided Instruction", Chapman and Hall, London.

Robinson, J.A. (1965) A machine-oriented logic based on the resolution principle Journal of the ACM 12, 1, (January), pp23-41.

Rousset, M.C. (1988) On the Consistency of Knowledge Bases: COVADIS System, ECAI88, Munich.

Sleeman, D.H. (1981) A rule-based task generation system Proceedings of the 7th International Joint Conference on Artificial Intelligence, Vancouver, pp882-886.

Sleeman, D.H. (1982) Inferring (mal) rules from pupil's protocols Proceedings of ECAI'82, Pisa.

Sleeman, D.H. (1983) Inferring student models for intelligent computer-aided instruction, in R. Michalski, J. Carbonell and T. Mitchell (eds) "Machine Learning" Tioga Publishing Co., Los Altos, California.

Sleeman, D.H. & Smith, M.J. (1981) Modelling Student's Problem Solving Artificial Intelligence, 16, pp171-188.

Stansfield, J., Carr, B. & Goldstein, I. (1976) Wumpus advisor I: a first implementation of a program that tutors logical an probabilistic reasoning skills MIT AI Lab Memo #381.

Steele, G.L., Jr. (1990) Common Lisp The Language, 2nd Edition, Digital Press.

Stevens, A. & Collins, A. (1977) The goal structure of a Socratic tutor Proceedings of the ACM Annual Conference.

Stickel, M.E. (1982) A nonclausal connection-graph resolution theorem-proving program Proceedings of the AAAI National Conference on Artificial Intelligence, Pittsburgh, PA, pp229-233.

Thibodeau, R., Just, M.A. & Carpenter, P.A. (1982) A Model of the Time Course and Content of Reading Cognitive Science, 6, pp157-203.

VanLehn, K. (1982) Bugs are not enough: empirical studies of bugs, impasses and repairs in procedural skills Journal of Mathematical Behaviour, 3, pp3-72.

Venken, R. (1984) A Prolog meta-interpreter for partial evaluation and its application to source transformation and query-optimisation Proceedings of ECAI-84, Pisa, Italy, pp91-100.

Warren, D.H.D. (1977) Logic programming and compiler writing DAI Research Report 44, University of Edinburgh.

Waters, R.A. (1979) A Method for Analyzing Loop Programs IEEE Transactions on Software Engineering, SE-5:3, May.

Wertz, H. (1982) Stereotyped program debugging: an aid for novice programmers International Journal of Man-Machine Studies, 16, pp379-392.

Young, R.M. & O'Shea, T. (1981) Errors in children's subtraction Cognitive Science, 5, 2, pp155-177.

Appendix I

BNF-like Definition of PGPS's Syntax

Rule definitions are of the following form (ellipses denote the fact that the preceding item can be repeated, bold items are non-terminals, those underlined are terminals, and items in italics are optional):

rule	::=	(define-rule **rulename** if **lhs** then **rhs**)
lhs	::=	**lhs-term** *ampersand&lhs-term...*
lhs-term	::=	**function-specification**
	::=	**negated-function-specification**
	::=	**working-memory-pattern**
	::=	**negated-pattern**
ampersand&lhs-term	::=	& **lhs-term**
function-specification	::=	(**function-symbol** *Sexp...*)
	::=	(~ **function-specification**)
function-symbol	::=	*symbol
	::=	≥
	::=	≤
	::=	≥=
	::=	≤=
	::=	=
working-memory-pattern	::=	(**Sexp** *Sexp...*)
negated-pattern	::=	(~ **working-memory-pattern**)
Sexp	::=	**list**
	::=	**symbol**
	::=	**number**
	::=	**var**
	::=	**dont-care-variable**
list	::=	(*Sexp...*)
dont-care-variable	::=	?
var	:=	?**symbol**
rhs	::=	**rhs-term** *ampersand&rhs-term...*
rhs-term	::=	(**pattern-or-function-specification...**)
	::=	**working-memory-pattern**
	::=	(*output **Sexp**)
	::=	(*delete **number**)
	::=	(*halt)
ampersand&rhs-term	::=	& **rhs-term**

pattern-or-function-specification	::=	**Sexp**
	::=	**function-specification**
rulename	::=	**symbol**

Appendix IIa

Glossary of Symbols

Logic

$\exists$	There exists.	For example, $\exists$(x,y) **EATS(x,y)**, should be read as: There exists an *x* and a *y* such that EATS(x,y) is true.
$\forall$	For all.	e.g. $\forall$(x) **P(x)**, means that P is true for all x.
$\neg$	Not.	e.g. $\neg$**OPEN(x)**, means that x is not open.
$\wedge$	And.	e.g. **4<5** $\wedge$ $\neg$**5<4**, reads: 4 is less than 5, and 5 is not less than 4.
$\vee$	Or.	e.g. **4<5** $\vee$ **4=5**, reads: 4 is less than or equal to 5.
$\supset$	Implies.	e.g. $\forall$(x) **EVEN(x)** $\supset$ $\neg$**ODD(x)**, reads as: For all *x*, *x* being even implies that it is not odd.
$\equiv$	Equivalent to.	e.g. $\forall$(x) **EVEN(x)** $\equiv$ $\neg$**ODD(x)**, reads as: For all *x*, saying that *x* is even is equivalent to saying that it is not odd.

Sets

$\{x_1,...,x_n\}$	The set of elements x_1 through x_n.	
$\in$	Is a member of.	For example, **apple**$\in$ **Fruits**, should be read as: Apple is a member of the set of fruits.
$\notin$	Is not a member of.	
$\cup$	Union.	e.g. **Apples**$\cup$**Pears**, reads as: The union of the sets of apples and pears.
|	Such that.	e.g. {x|x$\in$ **NATNUM**$\wedge$**EVEN(x)**}, reads as: The set of *x*s such that *x* is an even natural number.
$\times$	Cartesian Product.	The cartesian product of two sets, A and B, is: {(a,b)|a$\in$ A$\wedge$b$\in$ B}.
-	Minus.	e.g. A-B means the set of elements remaining after removing all of the members of B from A, or: {x|x$\in$ A$\wedge$x$\notin$ B}.
#	Cardinality.	e.g. #A is the number of elements in the set A.
$\sqsupset$	Order	e.g. x$\sqsupset$y reads as *x* is 'greater than' *y* in this ordering.

$\subseteq$	Subset.	e.g. **Apples$\subseteq$Fruits** means that the set of apples is a subset of the set of fruits.

Functions

$\rightarrow$	Maps.	e.g. **double: NUMBERS $\rightarrow$ EVENS**, reads: *Double* is a function which maps each member of the set of numbers to a member of the set of even numbers.

Appendix IIb

Glossary of Terms

Term	Section	Description
Abstract Inter-pattern Consistency	3.5.4	Abstract version of the Inter-pattern Consistency rule.
Abstract Negated Pattern Instantiation Set	3.5.4	A Negated Pattern Instantiation Set for abstract WMEs.
Abstract Pattern Instantiation Set	3.5.4	Analogous to a Pattern Instantiation Set, but pertains to a set of abstract WMEs.
Abstract Rule Instantiation	3.5.4	A rule instantiation which has been derived from a set of abstract WMEs.
Abstract WME-uniqueness	3.5.4	Like WME-uniqueness, but pertains to abstract WMEs.
Conflict Set	3.4.1	The set of current instantiations.
CP	1.3	A 'Critical Problem', i.e. an input which discriminates between to candidate models.
Exclusion Clause	3.7	A clause which defines the conditions under which a given rule would fire because a competitor, which would have won during conflict resolution, fails to be instantiated at all.
Input Specification	3.5.1	A formal description of the set of inputs which a set of production rules is intended to handle.
Inter-pattern Consistency	3.4.1	Rejects Preliminary Rule Instantiations where the inter-pattern bindings are inconsistent.
Negated Pattern Instantiation Set	3.4.1	Analogous to the Pattern Instantiation Set but pertains to negated patterns only.
Pattern Instantiation Set	3.4.1	For a given rule, the set of instantiations of the patterns in the LHS of the rule, with respect to the current state of WM.
Preliminary Abstract Rule Instantiation Set	3.5.4	Analogous to a Preliminary Rule Instantiation Set, but pertains to abstract data.

Term	Section	Description
Preliminary Rule Instantiation Set	3.4.1	Obtained by computing the Cartesian Product of the elements in the Pattern Instantiation Set. It ignores consistency issues between the patterns in each tuple; these are dealt with by applying the rules of WME-uniqueness and Inter-pattern Consistency.
PS Dependency Network	2.5.2	Graph representation of the dependencies between the LHSs and RHSs of a set of production rules.
Recency	3.4.2	This conflict resolution strategy favours instantiations which match more recent elements in WM.
Refractoriness	3.4.2	A conflict resolution strategy which excludes those instantiations which have fired on previous cycles of the interpreter.
Specificity	3.4.2	This conflict resolution strategy favours more specific instantiations.
WME-uniqueness	3.4.1	Rejects those Preliminary Rule Instantiations which multiply reference a given WME.

Appendix III

Traces of Chapter 2 Models

The traces of PG's analysis of the models presented in Chapter 2 can be found below. Section 4.1 describes how to interpret the output.

CP-search for rulesets SUBTRACT and ABS-SUBTRACT,
on Input Specs: (((?m - ?s)))
In the following trace, 'CS' denotes the Conflict Set, and the numbers 1, 2, and 3, on the same line, denote the application of Refractoriness, Recency, and Specificity, respectively.

Choosing Input Specification: ((?m.0 - ?s.0))

[1]
Context: SUBTRACT
Trying: negative-result...
Checking pattern-set consistency for #negative-result...
Satisfiables: NIL
Negated Theorem: LT(?m.0,?s.0)
User interrupt: ...Yes... is consistent.Trying: positive-result...
Checking pattern-set consistency for #positive-result...
Satisfiables: NIL
Negated Theorem: ¬LT(?m.0,?s.0)
User interrupt: ...Yes... is consistent.Trying: subtract... Trying: halt1...
CS: (#negative-result #positive-result), 1: (#negative-result #positive-result), 2: (#positive-result #negative-result), 3: (#negative-result #positive-result),
#negative-result has the following competitors: (#positive-result), yielding exclusion clauses: LT(?m.0,?s.0)
#positive-result has the following competitors: (#negative-result), yielding exclusion clauses: ¬LT(?m.0,?s.0)
Contradictory instantiations: NIL

[2]
Context: ABS-SUBTRACT
Trying: positive-result...
Checking pattern-set consistency for #positive-result...
Satisfiables: NIL
Negated Theorem: ¬LT(?m.0,?s.0) ... is consistent.Trying: swap-numbers...
Checking pattern-set consistency for #swap-numbers...
Satisfiables: NIL
Negated Theorem: LT(?m.0,?s.0) ... is consistent.Trying: halt2...
CS: (#positive-result #swap-numbers), 1: (#positive-result #swap-numbers), 2: (#swap-numbers #positive-result), 3: (#positive-result #swap-numbers),
#positive-result has the following competitors: (#swap-numbers), yielding exclusion clauses: ¬LT(?m.0,?s.0)
#swap-numbers has the following competitors: (#positive-result), yielding exclusion clauses: LT(?m.0,?s.0)
Contradictory instantiations: NIL

Checking consistency of State Pair...
Satisfiables: LT(?m.0,?s.0)
Negated Theorem: ¬LT(?m.0,?s.0)... is not consistent.
The pairing: SUBTRACT/#negative-result with ABS-SUBTRACT/#positive-result is inconsistent.
The pairing: SUBTRACT/#negative-result with ABS-SUBTRACT/#swap-numbers is consistent.
The pairing: SUBTRACT/#positive-result with ABS-SUBTRACT/#positive-result is consistent.
Checking consistency of State Pair...
Satisfiables: ¬LT(?m.0,?s.0)
Negated Theorem: LT(?m.0,?s.0) ... is not consistent.
The pairing: SUBTRACT/#positive-result with ABS-SUBTRACT/#swap-numbers is inconsistent.
(Initialisation Complete)

Selecting context: SUBTRACT

[1]Firing: #negative-result
Deposited: ((- ?s.0 - ?m.0))
WM: ((- ?s.0 - ?m.0) (?m.0 - ?s.0))
Constraints: LT(?m.0,?s.0)
Exclusions: LT(?m.0,?s.0)

[3]_____
Context: SUBTRACT
Trying: negative-result... Trying: positive-result...
Checking pattern-set consistency for #positive-result...
  Satisfiables: LT(?m.0,?s.0)
  Negated Theorem: ¬LT(?m.0,?s.0) ... is not consistent.Trying: subtract... Trying: halt1...
CS: (#negative-result #subtract), 1: (#subtract), 2: (#subtract), 3: (#subtract),
#subtract has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: SUBTRACT/#subtract with ABS-SUBTRACT/#swap-numbers is consistent.

[3]Firing: #subtract
Deposited: ((- (*subtract ?s.0 ?m.0)))
WM: ((- (*subtract ?s.0 ?m.0)) (- ?s.0 - ?m.0) (?m.0 - ?s.0))
Constraints: LT(?m.0,?s.0)
Exclusions: LT(?m.0,?s.0)

[4]_____
Context: SUBTRACT
Trying: negative-result... Trying: positive-result...
Checking pattern-set consistency for #positive-result...
  Satisfiables: LT(?m.0,?s.0)
  Negated Theorem: ¬LT(?m.0,?s.0) ... is not consistent.Trying: subtract... Trying: halt1...
CS: (#negative-result #subtract #halt1), 1: (#halt1), 2: (#halt1), 3: (#halt1),
#halt1 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: SUBTRACT/#halt1 with ABS-SUBTRACT/#swap-numbers is consistent.

[4]Firing: #halt1
WM: ((- (*subtract ?s.0 ?m.0)) (- ?s.0 - ?m.0) (?m.0 - ?s.0))
Constraints: LT(?m.0,?s.0)
Exclusions: LT(?m.0,?s.0)
New outputs: ((*output -) (*output (*subtract ?s.0 ?m.0)))
Halt signalled on this path.
Switching rulesets because of *HALT.

_____
Selecting context: ABS-SUBTRACT

[2]Firing: #swap-numbers
Deposited: ((+ ?s.0 - ?m.0))
WM: ((+ ?s.0 - ?m.0) (?m.0 - ?s.0))
Constraints: LT(?m.0,?s.0)
Exclusions: LT(?m.0,?s.0)

[5]_____
Context: ABS-SUBTRACT
Trying: positive-result...
Checking pattern-set consistency for #positive-result...
  Satisfiables: LT(?m.0,?s.0)
  Negated Theorem: ¬LT(?m.0,?s.0) ... is not consistent.Trying: swap-numbers... Trying: halt2...
Checking pattern-set consistency for #halt2...
  Satisfiables: LT(?m.0,?s.0)
  Negated Theorem: ¬LT(?s.0,?m.0)
  User interrupt: ...Yes... is consistent.
CS: (#swap-numbers #halt2), 1: (#halt2), 2: (#halt2), 3: (#halt2),
#halt2 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: SUBTRACT/#halt1 with ABS-SUBTRACT/#halt2 is consistent.

[5]Firing: #halt2
WM: ((+ ?s.0 - ?m.0) (?m.0 - ?s.0))
Constraints: (¬LT(?s.0,?m.0) ∧ LT(?m.0,?s.0))
Exclusions: LT(?m.0,?s.0)
New outputs: ((*output +) (*output (*subtract ?s.0 ?m.0)))
Halt signalled on this path.

_____
CP-generation info:
Input Spec: ((?m.0 - ?s.0))

State information for STATE1. Ruleset: SUBTRACT
 Instantiations: (#negative-result #subtract #halt1)
 Outputs: ((*output -) (*output (*subtract ?s.0 ?m.0)))
  LT(?m.0,?s.0)
Exclusion clauses: LT(?m.0,?s.0)
This ruleset is in a halt state.

State information for STATE2. Ruleset: ABS-SUBTRACT
 Instantiations: (#swap-numbers #halt2)
 Outputs: ((*output +) (*output (*subtract ?s.0 ?m.0)))
  (¬LT(?s.0,?m.0) ∧ LT(?m.0,?s.0))
Exclusion clauses: LT(?m.0,?s.0)
This ruleset is in a halt state.

The outputs - and + cannot be unified. Therefore, a problem, satisfying the above constraint set, will discriminate between the two models.

No CP can be generated for the outputs (*subtract ?s.0 ?m.0) and (*subtract ?s.0 ?m.0), as they are equal.
Switching rulesets because of *HALT.

Selecting context: SUBTRACT

[1]Firing: #positive-result
Deposited: ((+ ?m.0 - ?s.0))
WM: ((+ ?m.0 - ?s.0) (?m.0 - ?s.0))
Constraints: ¬LT(?m.0,?s.0)
Exclusions: ¬LT(?m.0,?s.0)

[6]
Context: SUBTRACT
Trying: negative-result...
Checking pattern-set consistency for #negative-result...
Satisfiables: ¬LT(?m.0,?s.0)
Negated Theorem: LT(?m.0,?s.0) ... is not consistent.Trying: positive-result... Trying: subtract... Trying: halt1...
CS: (#positive-result #subtract), 1: (#subtract), 2: (#subtract), 3: (#subtract),
#subtract has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: SUBTRACT/#subtract with ABS-SUBTRACT/#positive-result is consistent.

[6]Firing: #subtract
Deposited: ((+ (*subtract ?m.0 ?s.0)))
WM: ((+ (*subtract ?m.0 ?s.0)) (+ ?m.0 - ?s.0) (?m.0 - ?s.0))
Constraints: ¬LT(?m.0,?s.0)
Exclusions: ¬LT(?m.0,?s.0)

[7]
Context: SUBTRACT
Trying: negative-result...
Checking pattern-set consistency for #negative-result...
Satisfiables: ¬LT(?m.0,?s.0)
Negated Theorem: LT(?m.0,?s.0) ... is not consistent.Trying: positive-result... Trying: subtract... Trying: halt1...
CS: (#positive-result #subtract #halt1), 1: (#halt1), 2: (#halt1), 3: (#halt1),
#halt1 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: SUBTRACT/#halt1 with ABS-SUBTRACT/#positive-result is consistent.

[7]Firing: #halt1
WM: ((+ (*subtract ?m.0 ?s.0)) (+ ?m.0 - ?s.0) (?m.0 - ?s.0))
Constraints: ¬LT(?m.0,?s.0)
Exclusions: ¬LT(?m.0,?s.0)
New outputs: ((*output +) (*output (*subtract ?m.0 ?s.0)))
Halt signalled on this path.
Switching rulesets because of *HALT.

Selecting context: ABS-SUBTRACT

[2]Firing: #positive-result
Deposited: ((+ ?m.0 - ?s.0))
WM: ((+ ?m.0 - ?s.0) (?m.0 - ?s.0))
Constraints: ¬LT(?m.0,?s.0)
Exclusions: ¬LT(?m.0,?s.0)

[8]
Context: ABS-SUBTRACT
Trying: positive-result... Trying: swap-numbers...
Checking pattern-set consistency for #swap-numbers...
Satisfiables: ¬LT(?m.0,?s.0)
Negated Theorem: LT(?m.0,?s.0) ... is not consistent.Trying: halt2...
CS: (#positive-result #halt2), 1: (#halt2), 2: (#halt2), 3: (#halt2),
#halt2 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: SUBTRACT/#halt1 with ABS-SUBTRACT/#halt2 is consistent.

[8]Firing: #halt2
WM: ((+ ?m.0 - ?s.0) (?m.0 - ?s.0))
Constraints: ¬LT(?m.0,?s.0)
Exclusions: ¬LT(?m.0,?s.0)
New outputs: ((*output +) (*output (*subtract ?m.0 ?s.0)))
Halt signalled on this path.

CP-generation info:
Input Spec: ((?m.0 - ?s.0))

State information for STATE1. Ruleset: SUBTRACT
Instantiations: (#positive-result #subtract #halt1)
Outputs: ((*output +) (*output (*subtract ?m.0 ?s.0)))
¬LT(?m.0,?s.0)

Exclusion clauses: ¬LT(?m.0,?s.0)
This ruleset is in a halt state.

State information for STATE2. Ruleset: ABS-SUBTRACT
Instantiations: (#positive-result #halt2)
Outputs: ((*output +) (*output (*subtract ?m.0 ?s.0)))
¬LT(?m.0,?s.0)
Exclusion clauses: ¬LT(?m.0,?s.0)
This ruleset is in a halt state.

No CP can be generated for the outputs + and +, as they are equal.

No CP can be generated for the outputs (*subtract ?s.0 ?m.0) and (*subtract ?s.0 ?m.0), as they are equal.
Switching rulesets because of *HALT.

Selecting context: SUBTRACT
Switching ruleset because the current ruleset has no unhalted State Pairs.

Selecting context: ABS-SUBTRACT

No input specifications left.

CP-search for rulesets SIDE-BRANCH and NIL,

on Input Specs: (((go)))
In the following trace, 'CS' denotes the Conflict Set, and the numbers 1, 2, and 3, on the same line, denote the application of Refractoriness, Recency, and Specificity, respectively.

Choosing Input Specification: ((go))

[1]
Context: SIDE-BRANCH
Trying: start... Trying: side-branch... Trying: halt...
CS: (#start), 1: (#start), 2: (#start), 3: (#start),
#start has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

[2]
Context: NIL

CS: NIL, 1: NIL,
The pairing: SIDE-BRANCH/#start with NIL/NIL is consistent.
NIL is in a forced halt state.
(Initialisation Complete)

Selecting context: SIDE-BRANCH

[1]Firing: #start
Deposited: ((counter is 1))
WM: ((counter is 1) (go))
Constraints: NIL
Exclusions: NIL

[3]
Context: SIDE-BRANCH
Trying: start... Trying: side-branch... Trying: halt...
CS: (#start #side-branch #halt), 1: (#side-branch #halt), 2: (#halt #side-branch), 3: (#side-branch),
#side-branch has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL
2: (#halt), 3: (#halt),
#halt has the following competitors: (#side-branch), yielding exclusion clauses: (NIL)
Checking exclusion clauses for #halt...
Satisfiables: NIL
Negated Theorem: (NIL)... is not consistent.
Contradictory instantiations: (#halt)

The pairing: SIDE-BRANCH/#side-branch with NIL/NIL is consistent.
NIL is in a forced halt state.

[3]Firing: #side-branch
WM: ((counter is 1) (go))
Constraints: NIL
Exclusions: NIL
New outputs: ((*output 1))

[4]
Context: SIDE-BRANCH
Trying: start... Trying: side-branch... Trying: halt...
CS: (#start #side-branch #halt), 1: (#halt), 2: (#halt), 3: (#halt),
#halt has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: SIDE-BRANCH/#halt with NIL/NIL is consistent.

NIL is in a forced halt state.

[4]Firing: #halt
WM: ((counter is 1) (go))
Constraints: NIL
Exclusions: NIL
New outputs: ((*output (*add1 1)))
Halt signalled on this path.

CP-generation info:
Input Spec: ((go))

State information for STATE1. Ruleset: SIDE-BRANCH
Instantiations: (#start #side-branch #halt)
Outputs: ((*output 1) (*output (*add1 1)))
NIL
Exclusion clauses: NIL
This ruleset is in a halt state.

State information for STATE2. Ruleset: NIL
Instantiations: NIL
Outputs: NIL
This ruleset is in a halt state.

The models can be distinguished by the fact that they produce different numbers of outputs.
Switching rulesets because of *HALT.

Selecting context: NIL
Switching ruleset because the current ruleset has no unhalted State Pairs.

Selecting context: SIDE-BRANCH

No input specifications left.

CP-search for rulesets ONE-TWO1 and ONE-TWO2,

on Input Specs: (((start)))
In the following trace, 'CS' denotes the Conflict Set, and the numbers 1, 2, and 3, on the same line, denote the application of Refractoriness, Recency, and Specificity, respectively.

Choosing Input Specification: ((start))

[1]
Context: ONE-TWO1
Trying: r1... Trying: r2... Trying: r3...
CS: (#r1), 1: (#r1), 2: (#r1), 3: (#r1),
#r1 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

[2]
Context: ONE-TWO2
Trying: r2... Trying: r3... Trying: r1-swapped...
CS: (#r1-swapped), 1: (#r1-swapped), 2: (#r1-swapped), 3: (#r1-swapped),
#r1-swapped has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: ONE-TWO1/#r1 with ONE-TWO2/#r1-swapped is consistent.
(Initialisation Complete)

Selecting context: ONE-TWO1

[1]Firing: #r1
Deposited: ((first action) (second action))
WM: ((second action) (first action) (start))
Constraints: NIL
Exclusions: NIL

[3]
Context: ONE-TWO1
Trying: r1... Trying: r2... Trying: r3...
CS: (#r1 #r2 #r3), 1: (#r2 #r3), 2: (#r2), 3: (#r2),
#r2 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL
2: (#r3), 3: (#r3),
#r3 has the following competitors: (#r2), yielding exclusion clauses: (NIL)
Checking exclusion clauses for #r3...
Satisfiables: NIL
Negated Theorem: (NIL)... is not consistent.
Contradictory instantiations: (#r3)

The pairing: ONE-TWO1/#r2 with ONE-TWO2/#r1-swapped is consistent.

[3]Firing: #r2
WM: ((second action) (first action) (start))
Constraints: NIL
Exclusions: NIL
New outputs: ((*output 1))

[4]_____
Context: ONE-TWO1
Trying: r1... Trying: r2... Trying: r3...
CS: (#r1 #r2 #r3), 1: (#r3), 2: (#r3), 3: (#r3),
#r3 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: ONE-TWO1/#r3 with ONE-TWO2/#r1-swapped is consistent.

[4]Firing: #r3
WM: ((second action) (first action) (start))
Constraints: NIL
Exclusions: NIL
New outputs: ((*output 2))
Halt signalled on this path.
Switching rulesets because of *HALT.

_____
Selecting context: ONE-TWO2

[2]Firing: #r1-swapped
Deposited: ((second action) (first action))
WM: ((first action) (second action) (start))
Constraints: NIL
Exclusions: NIL

[5]_____
Context: ONE-TWO2
Trying: r2... Trying: r3... Trying: r1-swapped...
CS: (#r2 #r3 #r1-swapped), 1: (#r2 #r3), 2: (#r3), 3: (#r3),
#r3 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL
2: (#r2), 3: (#r2),
#r2 has the following competitors: (#r3), yielding exclusion clauses: (NIL)
Checking exclusion clauses for #r2...
Satisfiables: NIL
Negated Theorem: (NIL)... is not consistent.
Contradictory instantiations: (#r2)

The pairing: ONE-TWO1/#r3 with ONE-TWO2/#r3 is consistent.

[5]Firing: #r3
WM: ((first action) (second action) (start))
Constraints: NIL
Exclusions: NIL
New outputs: ((*output 2))
Halt signalled on this path.

_____
CP-generation info:
Input Spec: ((start))

State information for STATE1. Ruleset: ONE-TWO1
Instantiations: (#r1 #r2 #r3)
Outputs: ((*output 1) (*output 2))
NIL
Exclusion clauses: NIL
This ruleset is in a halt state.

State information for STATE2. Ruleset: ONE-TWO2
Instantiations: (#r1-swapped #r3)
Outputs: ((*output 2))
NIL
Exclusion clauses: NIL
This ruleset is in a halt state.

The models can be distinguished by the fact that they produce different numbers of outputs.
Switching rulesets because of *HALT.

_____
Selecting context: ONE-TWO1
Switching ruleset because the current ruleset has no unhalted State Pairs.

_____
Selecting context: ONE-TWO2

No input specifications left.

Appendix IVa

The Fraction Subtraction Rules

The set of rules, beginning with **WN1** and ending with **HLT**, are for solving fraction subtraction problems using the 'mixed' algorithm. Subsequent rules are 'mal-rules'. Each rule is annotated, and includes the frequency with which it appeared in the original 29 models (e.g. (f=2)).

```
(define-rule wn1
 #| The first whole number is loose, so cross it out, decrement it, and deposit a new fractional part DY/DY, where DY is the
denominator of the second fraction. (f=5)|#
 if ((wn 1) ?w) & ((fr 2) ? / ?dy) & ~((fr 1) ? / ?)
 then (*delete 1) & ((fr 1) ?dy / ?dy) & ((wn 1) (*dec ?w)))

(define-rule wn2
 #| The second whole number is loose, so just write down the first fraction. (f=20)|#
 if ((wn 2) ?) & ((fr 1) ?nx / ?dx) & ~((fr 2) ? / ?)
 then (?nx /) & (/ ?dx))

(define-rule wn3
 #| There is no whole number, so just deposit the first whole number. (f=20)|#
 if ((wn 1) ?w) & ~((wn 2) ?)
 then (?w))

(define-rule wn4
 #| Subtract the whole numbers if WN1 is mixed. If it is not mixed, then you will need to borrow from WN1 before
subtracting the whole numbers. (f=21)|#
 if ((wn 1) ?wx) & ((wn 2) ?wy) & ((fr 1) ? / ?)
 then ((*subtract ?wx ?wy)))

(define-rule nm1
 #| Having calculated the LCM, the second numerator is greater than the first, so cross-out the wole number (if not zero)
and decrement it, depositing a signal telling the borrow-rule to enlarge the numerator. (f=8)|#
 if (/ ?d) & ((fr 1) ?nx / ?d) & ((fr 2) ?ny / ?d) & (?w) & (< ?nx ?ny) & ~(= ?w 0)
 then ((*dec ?w)) & (enlarge ?nx) & (*delete 4))

(define-rule nm2a
 #| Subtracts the numerators if the denominator has been calculated and the minuend's numerator is greater than that of
the subtrahend. (f=18)|#
 if (/ ?d) & ((fr 1) ?nx / ?d) & ((fr 2) ?ny / ?d) & (> ?nx ?ny)
 then ((*subtract ?nx ?ny) /))

(define-rule nm2b
 #| If the denominator has been calculated and the numerators are equal, then the new numerator is just zero. (f=18)|#
 if (/ ?d) & ((fr 1) ?nx / ?d) & ((fr 2) ?nx / ?d)
 then (0 /))

(define-rule brw
 #| To borrow, add the denominator to the numerator. (f=6)|#
 if (enlarge ?nx) & (/ ?d)
 then ((fr 1) (*add ?d ?nx) / ?d))

(define-rule dn1
 #| The denominators are equal, so just deposit their value. (f=25)|#
 if ((fr 1) ? / ?d) & ((fr 2) ? / ?d) & ~(/ ?)
 then (/ ?d))

(define-rule dn2
 #| The denominators are not equal, therefore compute their LCM. (f=20)|#
 if ((fr 1) ? / ?dx) & ((fr 2) ? / ?dy) & ~(= ?dx ?dy) & ~(/ ?)
 then (make fractions alike ?dx ?dy))

(define-rule lcm
 #| Calculate the LCM of the two denominators and deposit in WM. (f=16)|#
 if (make fractions alike ?dx ?dy)
 then (/ (*findlcm ?dx ?dy)))
```

```
(define-rule fct1
  #| If the goal is to make the denominators of the fractions alike, the denominator is known but the new numerator of
the minuend has not been calculated, then work out the factor for the numerator by dividing the new denominator by
the old one. (f=13)|#
  if (make fractions alike ? ?) & ((fr 1) ? / ?olddn) & (/ ?d) & ~(new (fr 1) numerator ?)
  then ((fr 1) factor (*divide ?d ?olddn)))

(define-rule fct2
  #| Having worked out the new numerator of the minuend, calculate the factor for the subtrahend by dividing the new
denominator by the old one. (f=13)|#
  if ((fr 2) ? / ?olddn) & (/ ?d) & (new (fr 1) numerator ?)
  then ((fr 2) factor (*divide ?d ?olddn)))

(define-rule fct3
  #| If the new numerator for fractin ?x has not been calculated, then do so by multiplying the computed factor by
the old numerator. (f=13)|#
  if ((fr ?x) factor ?fct) & ((fr ?x) ?n / ?) & ~(new (fr ?x) numerator ?)
  then (new (fr ?x) numerator (*multiply ?fct ?n)))

(define-rule fct4
  #| Having calculated the new numerators and denominator, use them to make up the new fractions. (f=13)|#
  if (new (fr 1) numerator ?nx) & (new (fr 2) numerator ?ny) & (/ ?d)
  then ((fr 1) ?nx / ?d) & ((fr 2) ?ny / ?d))

(define-rule hcf
  #| If the numerator is not zero, the denominator is greater than the numerator and a coancel operation has not been
performed, then try cancelling and note that the attempt has been made. (f=11)|#
  if (?n /) & ~(= ?n 0) & (/ ?d) & (*needs-cancelling-p ?n ?d) & (> ?d ?n)
     & ~(have cancelled)
  then (*delete 1) & (*delete 2) & ((*cancel-nm ?n ?d) /) & (/ (*cancel-dn ?n ?d)) & (have cancelled))

(define-rule end1
  #| Having calculated the numerator and denominator, we now have part of the final answer. (f=29)|#
  if (?n /) & ~(= ?n 0) & (/ ?d)
  then (answer is (?n / ?d)))

(define-rule end2
  #| If the answer has also got a whole number component, then include it. (f=28)|#
  if (?w) & ~(= ?w 0) & (answer is (?n / ?d))
  then (answer is (?w and ?n / ?d)))

(define-rule end3
  #| A null numerator means that the answer is zero. When a whole number is present, this rule is over-ridden by END4. (f=6)|#
  if (0 /)
  then (answer is (0)))

(define-rule end4
  #| If the answer includes a whole number and the numerator is null, then discard the fractional part from the answer. (f=2)|#
  if (?w) & (0 /)
  then (answer is (?w)))

(define-rule hlt
  #| All finished, so output the final answer and halt execution. (f=29)|#
  if (answer is ?ans)
  then (*output ?ans) & (*halt))

(define-rule wn1m
  #| If the first whole number is loose, then deposit the fractional part of the second fraction. (f=5)|#
  if ((wn 1) ?wx) & ((wn 2) ?wy) & ((fr 2) ?n / ?d) & ~((fr 1) ? / ?)
  then ((*subtract ?wx ?wy)) & (?n /) & (/ ?d))

(define-rule wn4m
  #| Gets invoked once the mal-rule, PG1, has successfully discarded the fractional parts. This rule deposits the difference
of the two whole numbers. (f=1)|#
  if ((wn 1) ?wx) & ((wn 2) ?wy)
  then ((*abs-subtract ?wx ?wy)))

(define-rule nm2m1
  #| Having caluclated the denominator, take the absolute difference of the two numerators. (f=3)|#
  if (/ ?) & ((fr 1) ?nx / ?) & ((fr 2) ?ny / ?)
  then ((*abs-subtract ?nx ?ny) /))

(define-rule nm2m2
  #| Deposits the absolute difference between the two numerators regardless of whether the denominators are equal or not. (f=6)|#
  if ((fr 1) ?nx / ?) & ((fr 2) ?ny / ?)
  then ((*abs-subtract ?nx ?ny) /))

(define-rule nm2m3a
  #| Subtract the numerators without first calculating the common denominator provided that the minuend numerator is
greater than that of the subtrahend. (f=1)|#
  if ((fr 1) ?nx / ?) & ((fr 2) ?ny / ?) & (< ?ny ?nx)
  then ((*subtract ?nx ?ny) /))

(define-rule nm2m3b
  #| Subtract the numerators without first calculating the common denominator provided that the minuend numerator is
equal to that of the subtrahend. (f=1)|#
  if ((fr 1) ?n / ?) & ((fr 2) ?n / ?)
  then (0 /))
```

```
(define-rule brwm
 #| When borrowing add ten to the numerator. (f=2)|#
 if (enlarge ?nx) & (/ ?d)
 then ((fr 1) (*add 10 ?nx) / ?d))

(define-rule dn1v
 #| This rule sets up the goal of calculating the common denominator, even thought the denominators are already equal. (f=1)|#
 if ((fr 1) ? / ?d) & ((fr 2) ? / ?d) & ~(/ ?)
 then (/ ?d) & (make fractions alike ?d ?d))

(define-rule lcmv
 #| Does not calculate the LCM. Rather, it returns a common denominator (by multiplying the two
denominators together). Provided that it is coupled with the three FCTv rules, it will lead to the correct answer. (f=4)|#
 if (make fractions alike ?dx ?dy)
 then (/ (*multiply ?dx ?dy)))

(define-rule fctv1
 #| If the goal is to make the fractions alike and the new minuend numerator has not been calculated, then multiply the
old minuend numerator by the denominator of the subtrahend. (f=6)|#
 if (/ ?) & (make fractions alike ? ?fct) & ((fr 1) ?nx / ?) & ~(new (fr 1) numerator ?)
 then (new (fr 1) numerator (*multiply ?fct ?nx)))

(define-rule fctv2
 #| If the goal is to make the fractions alike and the new subtrahend numerator has not been calculated, then multiply
the old subtrahend numerator by the denominator of the minuend. (f=6)|#
 if (new (fr 1) numerator ?) & (make fractions alike ?fct ?) & ((fr 2) ?ny / ?) & ~(new (fr 2) numerator ?)
 then (new (fr 2) numerator (*multiply ?fct ?ny)))

(define-rule fctv3
 #| Having calculated the numerators and denominator, use them to make up the new fractions (f=6)|#
 if (new (fr 1) numerator ?nx) & (new (fr 2) numerator ?ny) & (/ ?d)
 then ((fr 1) ?nx / ?d) & ((fr 2) ?ny / ?d))

(define-rule dn2m1
 #| Deposit the absolute difference of the two denominators, if they are not equal. (f=7)|#
 if (? /) & ((fr 1) ? / ?dx) & ((fr 2) ? / ?dy) & ~(= ?dx ?dy)
 then (/ (*abs-subtract ?dx ?dy)))

(define-rule dn2m2
 #| Only deposit the difference of the two denominators if the minuend's denominator is greater than that of the subtrahend. (f=1)|#
 if (? /) & ((fr 1) ? / ?dx) & ((fr 2) ? / ?dy) & (< ?dy ?dx)
 then (/ (*subtract ?dx ?dy)))

(define-rule end3m1
 #| If the computed numerator is zero, then change it back to that of the minuend or subtrahend
(they are both equal, so either will do). (f=7)|#
 if (0 /) & ((fr 1) ?n / ?) & ((fr 2) ?n / ?)
 then (?n /))

(define-rule end3m2
 #| Change 0/D to D (e.g. 0/2 becomes 2). (f=3)|#
 if (0 /) & (/ ?d)
 then (answer is (?d)))

(define-rule end4m1
 #| If the computed whole number is zero, then change it back to that of the minuend. (f=3)|#
 if (0) & ((wn 1) ?w)
 then (?w))

(define-rule end4m2
 #| Ignore the fractional part of the answer. (f=1)|#
 if (?w)
 then (answer is ?w))

(define-rule pg1
 #| This rule has been specially created as a companion to wn4m. It modifies the state of WM so that wn4m can fire.|#
 if ((wn 1) ?wx) & ((wn 2) ?wy) & (> ?wx ?wy)
 then (*delete 2) & ((wn 2) (*add1 ?wx)))
```

Appendix IVb

The Fraction Subtraction Models

These are the 'correct' and 'error' models used to evaluate PG. Each model definition is of the form: (DEFMODEL <model-name> <rulenames> RULE-SET). Each error model is preceded by the Lisp form which was used to evaluate it against a 'correct' model. On each run, the axioms and cached clauses are reinitialised (i.e. initialised to the empty set). PG is then provided with the axioms shown in each example. The search is then invoked via a call to Gimme-CP, which takes the two models, a set of Input Specifications, a start flag and a comment.

```
(DEFMODEL CORRECT-w1&2
(wn1 wn3 wn4 nm2a nm2b dn1 end1 end2 hlt)
RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
   (AX ∀(x,y) ¬EQ(x,y) ≡ LT(x,y) v LT(y,x))
   (AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0))
   (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
   (Gimme-CP 'CORRECT-w1&2 'ERROR-w1
     '((((wn 1) ?wx) ((fr 2) ?ny / ?dy) (< ?ny ?dy))) T
     "the child gives up with problems where the minuend is a loose whole number."))

(DEFMODEL ERROR-w1
(nm2a nm2b dn1 end1 end2 hlt)
RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
   (AX ∀(x,y) ¬EQ(x,y) ≡ LT(x,y) v LT(y,x))
   (AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0))
   (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
   (Gimme-CP 'CORRECT-w1&2 'ERROR-w2
         '((((wn 1) ?wx) ((wn 2) ?wy) ((fr 2) ?ny / ?dy) (< ?wy ?wx) (< ?ny ?dy))) T
        "the child writes down the fractional part of the second, if the first whole number is loose."))

(DEFMODEL ERROR-w2
(wn1m nm2a nm2b dn1 end1 end2 hlt)
RULE-SET)

(DEFMODEL CORRECT-w3
(wn2 wn4 nm2a nm2b dn1 end1 end2 hlt)
RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
   (AX ∀(x,y) ¬EQ(x,y) ≡ LT(x,y) v LT(y,x))
   (AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0))
   (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
   (Gimme-CP 'CORRECT-w3 'ERROR-w3
         '((((wn 1) ?wx) ((wn 2) ?wy) ((fr 1) ?nx / ?dx) (<= ?wy ?wx) (< ?nx ?dx))) T
        "the child gives up with problems where the subtrahend is a loose whole number."))

(DEFMODEL ERROR-w3
(nm2a nm2b dn1 end1 end2 hlt)
RULE-SET)

(DEFMODEL CORRECT-w4
(wn3 nm2a nm2b dn1 end1 end2 hlt)
RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
   (AX ∀(x,y) ¬EQ(x,y) ≡ LT(x,y) v LT(y,x))
   (AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0))
   (AX ∀(x) EQ(x,x)) ;Reflexivity
   (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
   (Gimme-CP 'CORRECT-w4 'ERROR-w4
         '((((wn 1) ?wx) ((fr 1) ?nx / ?dx) ((fr 2) ?ny / ?dy) (< ?nx ?dx) (< ?ny ?dy))) T
        "the child cannot deal with mixed fraction problems where only the first parameter is mixed."))
(DEFMODEL ERROR-w4
(nm2a nm2b dn1 end1 end2 hlt)
RULE-SET)
```

```
(DEFMODEL CORRECT-w5
 (wn4 nm2a dn1 end1 end2 hlt)
 RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
   (AX ∀(x,y) ¬EQ(x,y) ≡ LT(x,y) v LT(y,x))
   (AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0))
   (AX ∀(x) EQ(x,x)) ;Reflexivity
   (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
   (AX ∀(x) ¬LT(add1(x),x))
   (Gimme-CP 'CORRECT-w5 'ERROR-w5
         '((((wn 1) ?wx) ((wn 2) ?wy) ((fr 1) ?nx / ?dx) ((fr 2) ?ny / ?dy)
           (<= ?wy ?wx) (< ?nx ?dx) (< ?ny ?dy))) T
         "if after some other mistake the 1st whole number is less than the 2nd, then take the absolute
          difference.  The rule PG1 is included to force this error."))

(DEFMODEL ERROR-w5
 (pg1 wn4m nm2a dn1 end1 end2 hlt)
 RULE-SET)

(DEFMODEL CORRECT-n1&2&5
 (wn4 nm1 nm2a nm2b brw dn1 end1 end2 hlt)
 RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
   (AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0))
   (AX ∀(x) EQ(x,x)) ;Reflexivity
   (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
   (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
   (AX ∀(x,y) LT(x,y) ≡ ¬LT(y,x))  Asymmetric
   (Gimme-CP 'CORRECT-n1&2&5 'ERROR-n1
         '((((wn 1) ?wx) ((wn 2) ?wy) ((fr 1) ?nx / ?dx) ((fr 2) ?ny / ?dy)
           (< ?wy ?wx) (< ?nx ?ny) (< ?nx ?dx) (< ?ny ?dy))) T
         "If the problem requires a borrow, then get stuck."))

(DEFMODEL ERROR-n1
 (wn4 nm2a nm2b brw dn1 end1 end2 hlt)
 RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
   (AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0))
   (AX ∀(x) EQ(x,x)) ;Reflexivity
   (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
   (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
   (AX ∀(x,y) LT(x,y) ≡ ¬LT(y,x))  Asymmetric
   (Gimme-CP 'CORRECT-n1&2&5 'ERROR-n2
         '((((wn 1) ?wx) ((wn 2) ?wy) ((fr 1) ?nx / ?dx) ((fr 2) ?ny / ?dy)
           (< ?wy ?wx) (< ?nx ?ny) (= ?dx ?dy) (< ?nx ?dx) (< ?ny ?dy))) T
         "the child takes the absolute difference between the two numerators."))

(DEFMODEL ERROR-n2
 (wn4 nm2m1 brw dn1 end1 end2 hlt)
 RULE-SET)

(DEFMODEL CORRECT-n3&4-d1-f1&2
 (nm2a nm2b dn2 lcm fct1 fct2 fct3 fct4 end1 hlt)
 RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
   (AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0))
   (AX ∀(x) EQ(x,x)) ;Reflexivity
   (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
   (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
   (AX ∀(x,y) LT(x,y) ≡ ¬LT(y,x))  Asymmetric
   (AX ∀(a,b,c,d) LT(div(c,d),div(a,b))
     ⊃ ¬EQ(mult(div(findlcm(b,d),b),a),mult(div(findlcm(b,d),d),c)))
   (AX ∀(a,b,c,d) LT(a,c) ∧ EQ(d,findlcm(b,d))
     ⊃ ¬EQ(c,mult(div(findlcm(b,d),b),a)))
   (Gimme-CP 'CORRECT-n3&4-d1-f1&2 'ERROR-n3&d2
         '((((fr 1) ?nx / ?dx) ((fr 2) ?ny / ?dy) (< ?nx ?ny) (< ?dx ?dy)
           (< ?nx ?dx) (< ?ny ?dy) (< (*div ?ny ?dy) (*div ?nx ?dx)))) T
         "take the absolute difference between the two numerators before dealing with
          the denominators.  If the denominators are not equal then take their absolute difference."))

(DEFMODEL ERROR-n3&d2
 (nm2m2 dn2m1 end1 hlt)
 RULE-SET)
```

```
(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
    (AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0))
    (AX ∀(x) EQ(x,x)) ;Reflexivity
    (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
    (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
    (AX ∀(x,y) LT(x,y) ≡ ¬LT(y,x))  Asymmetric
    (AX ∀(a,b,c,d) LT(div(c,d),div(a,b))
      ⊃ ¬EQ(mult(div(findlcm(b,d),b),a),mult(div(findlcm(b,d),d),c)))
    (AX ∀(a,b,c,d) LT(a,c) ∧ EQ(d,findlcm(b,d))
      ⊃ ¬EQ(c,mult(div(findlcm(b,d),b),a)))
    (Gimme-CP 'CORRECT-n3&4-d1-f1&2 'ERROR-n4&d3
          '((((fr 1) ?nx / ?dx) ((fr 2) ?ny / ?dy) (< ?ny ?nx) (< ?dy ?dx) (< ?nx ?dx) (< ?ny ?dy)
            (< (*div ?ny ?dy) (*div ?nx ?dx)))) T
          "subtract numerators before denominators.  Subtract the two denominators."))

(DEFMODEL ERROR-n4&d3
 (nm2m3a nm2m3b dn2m2 end1 hlt)
 RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
    (AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0))
    (AX ∀(x) EQ(x,x)) ;Reflexivity
    (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
    (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
    (AX ∀(x,y) LT(x,y) ≡ ¬LT(y,x))  Asymmetric
    (Gimme-CP 'CORRECT-n1&2&5 'ERROR-n5
          '((((wn 1) ?wx) ((wn 2) ?wy) ((fr 1) ?nx / ?dx) ((fr 2) ?ny / ?dy) (< ?wy ?wx) (< ?nx ?ny)
            (= ?dx ?dy) (< ?nx ?dx) (< ?ny ?dy) (< ?nx 10) (< ?ny 10))) T
          "when borrowing, add ten to the numerator."))

(DEFMODEL ERROR-n5
 (wn4 nm1 nm2a nm2b brwm dn1 end1 end2 hlt)
 RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
    (AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0))
    (AX ∀(x) EQ(x,x)) ;Reflexivity
    (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
    (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
    (AX ∀(x,y) LT(x,y) ≡ ¬LT(y,x))  Asymmetric
    (AX ∀(a,b,c,d) LT(div(c,d),div(a,b))
      ⊃ ¬EQ(mult(div(findlcm(b,d),b),a),mult(div(findlcm(b,d),d),c)))
    (AX ∀(a,b,c,d) LT(a,c) ∧ EQ(d,findlcm(b,d))
      ⊃ ¬EQ(c,mult(div(findlcm(b,d),b),a)))
    (Gimme-CP 'CORRECT-n3&4-d1-f1&2 'ERROR-d1
          '((((fr 1) ?nx / ?dx) ((fr 2) ?ny / ?dy) (¬ (= ?dx ?dy)) (< ?nx ?dx) (< ?ny ?dy)
            (< (*div ?ny ?dy) (*div ?nx ?dx)))) T
          "the child cannot calculate the LCM."))

(DEFMODEL ERROR-d1
 (nm2a nm2b lcm fct1 fct2 fct3 fct4 end1 hlt)
 RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
    (AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0))
    (AX ∀(x) EQ(x,x)) ;Reflexivity
    (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
    (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
    (AX ∀(x,y) LT(x,y) ≡ ¬LT(y,x))  Asymmetric
    (AX ∀(a,b,c,d) LT(div(c,d),div(a,b))
      ⊃ ¬EQ(mult(div(findlcm(b,d),b),a),mult(div(findlcm(b,d),d),c)))
    (AX ∀(a,b,c,d) LT(a,c) ∧ EQ(d,findlcm(b,d))
      ⊃ ¬EQ(c,mult(div(findlcm(b,d),b),a)))
    (Gimme-CP 'CORRECT-n3&4-d1-f1&2 'ERROR-f1
          '((((fr 1) ?nx / ?dx) ((fr 2) ?ny / ?dy) (¬ (= ?dx ?dy)) (< ?nx ?dx) (< ?ny ?dy)
            (< (*div ?ny ?dy) (*div ?nx ?dx)))) T
          "work out the LCM but do not enlarge the numerators."))

(DEFMODEL ERROR-f1
 (nm2a nm2b dn2 lcm end1 hlt)
 RULE-SET)
```

```
(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
    (AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0))
    (AX ∀(x) EQ(x,x)) ;Reflexivity
    (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
    (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
    (AX ∀(x,y) LT(x,y) ≡ ¬LT(y,x))  Asymmetric
    (AX ∀(a,b,c,d) LT(div(c,d),div(a,b))
      ⊃ ¬EQ(mult(div(findlcm(b,d),b),a),mult(div(findlcm(b,d),d),c)))
    (AX ∀(a,b,c,d) LT(a,c) ∧ EQ(d,findlcm(b,d))
      ⊃ ¬EQ(c,mult(div(findlcm(b,d),b),a)))
    (Gimme-CP 'CORRECT-n3&4-d1-f1&2 'ERROR-f2
          '((((fr 1) ?nx / ?dx) ((fr 2) ?ny / ?dy) (¬ (= ?dx ?dy)) (< ?nx ?dx) (< ?ny ?dy)
            (< (*div ?ny ?dy) (*div ?nx ?dx)))) T
        "work out the LCM but enlarge the numerators by cross-multiplying."))

(DEFMODEL ERROR-f2
 (nm2a nm2b dn2 lcm fctv1 fctv2 fctv3 end1 hlt)
 RULE-SET)

(DEFMODEL CORRECT-f3
 (nm2a nm2b dn1 dn2 lcm fct1 fct2 fct3 fct4 end1 hlt)
 RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
    ;;(AX ∀(x,y) EQ(x,y) ≡ EQ(sub(x,y),0)) ;Sends into loop.
    (AX ∀(x) EQ(x,x)) ;Reflexivity
    (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
    (AX ∀(x,y,z) EQ(x,y) ∧ EQ(y,z) ⊃ EQ(x,z)) ;Transitivity
    (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
    (AX ∀(x,y) LT(x,y) ≡ ¬LT(y,x))  Asymmetric
    (Gimme-CP 'CORRECT-f3 'ERROR-f3
          '((((fr 1) ?nx / ?d) ((fr 2) ?ny / ?d) (< ?nx ?d) (< ?ny ?d))) T
        "factor even when the two denominators are equal."))

(DEFMODEL ERROR-f3
 (nm2a nm2b dn1v dn2 lcm fctv1 fctv2 fctv3 end1 hlt)
 RULE-SET)

(DEFMODEL CORRECT-h1
 (nm2a nm2b dn1 hcf end1 hlt)
 RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
    (AX ∀(x) EQ(x,x)) ;Reflexivity
    (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
    (AX ∀(x,y,z) EQ(x,y) ∧ EQ(y,z) ⊃ EQ(x,z)) ;Transitivity
    (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
    (Gimme-CP 'CORRECT-h1 'ERROR-h1
          '((((fr 1) ?nx / ?d) ((fr 2) ?ny / ?d) (< ?nx ?d) (< ?ny ?d))) T
        "do not cancel."))

(DEFMODEL ERROR-h1
 (nm2a nm2b dn1 end1 hlt)
 RULE-SET)

(DEFMODEL CORRECT-e1&2&3
 (nm2a nm2b dn1 end1 end3 hlt)
 RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
    (AX ∀(x) EQ(x,x)) ;Reflexivity
    (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
    (AX ∀(x,y,z) EQ(x,y) ∧ EQ(y,z) ⊃ EQ(x,z)) ;Transitivity
    (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
    (Gimme-CP 'CORRECT-e1&2&3 'ERROR-e1
          '((((fr 1) ?nx / ?d) ((fr 2) ?ny / ?d) (< ?nx ?d) (< ?ny ?d))) T
        "do not change 0/D to zero."))

(DEFMODEL ERROR-e1
 (nm2a nm2b dn1 end1 hlt)
 RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
    (AX ∀(x) EQ(x,x)) ;Reflexivity
    (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
    (AX ∀(x,y,z) EQ(x,y) ∧ EQ(y,z) ⊃ EQ(x,z)) ;Transitivity
    (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
    (Gimme-CP 'CORRECT-e1&2&3 'ERROR-e2
          '((((fr 1) ?nx / ?d) ((fr 2) ?ny / ?d) (< ?nx ?d) (< ?ny ?d))) T
        "change 0/D to N/D."))
```

```
(DEFMODEL ERROR-e2
(nm2a nm2b dn1 end1 end3m1 hlt)
RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
   (AX ∀(x) EQ(x,x)) ;Reflexivity
   (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
   (AX ∀(x,y,z) EQ(x,y) ∧ EQ(y,z) ⊃ EQ(x,z)) ;Transitivity
   (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
   (Gimme-CP 'CORRECT-e1&2&3 'ERROR-e3
          '((((fr 1) ?nx / ?d) ((fr 2) ?ny / ?d) (< ?nx ?d) (< ?ny ?d))) T
         "change 0/D to D."))

(DEFMODEL ERROR-e3
(nm2a nm2b dn1 end1 end3m2 hlt)
RULE-SET)

(DEFMODEL CORRECT-e4&5
(wn4 nm2a nm2b dn1 end1 end2 end4 hlt)
RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
   (AX ∀(x) EQ(x,x)) ;Reflexivity
   (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
   (AX ∀(x,y,z) EQ(x,y) ∧ EQ(y,z) ⊃ EQ(x,z)) ;Transitivity
   (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
   (Gimme-CP 'CORRECT-e4&5 'ERROR-e4
          '((((wn 1) ?w) ((wn 2) ?w) ((fr 1) ?nx / ?d) ((fr 2) ?ny / ?d) (< ?nx ?d) (< ?ny ?d))) T
         "change a whole number from 0 to W."))

(DEFMODEL ERROR-e4
(wn4 nm2a nm2b dn1 end1 end2 end4m1 hlt)
RULE-SET)

(progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
   (AX ∀(x) EQ(x,x)) ;Reflexivity
   (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
   (AX ∀(x,y,z) EQ(x,y) ∧ EQ(y,z) ⊃ EQ(x,z)) ;Transitivity
   (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
   (Gimme-CP 'CORRECT-e4&5 'ERROR-e5
          '((((wn 1) ?wx) ((wn 2) ?wy) ((fr 1) ?nx / ?d) ((fr 2) ?ny / ?d)
            (< ?nx ?d) (< ?ny ?d) (< ?wy ?wx))) T
         "leave the whole number out of the answer."))

(DEFMODEL ERROR-e5
(wn4 nm2a nm2b dn1 end1 end4 hlt)
RULE-SET)
```

Appendix IVc

CP-generation Output for the 20 Fraction Subtraction Models

This appendix presents the CP-generation information generated by PG for each of the error types, described in section 4.3. Section 4.1 describes how to interpret the output.

Error type W1

This model gives up with problems where the minuend is a loose whole number. PG correctly deduces that this model's behaviour differs from that of the correct one in the fact that it produces no output. There are two solution paths taken by the correct model. The first is followed when ¬EQ(dec(?wx),0); the second path is followed when the reverse is true. Note that the CP descriptions contain some redundant constraints, which could be removed by simplification. For example, ¬EQ(sub(?dy,?ny),0) ≡ ¬EQ(?dy,?ny), thus one of these constraints could be dropped with no loss of information. The first CP description could be instantiated by $2 - {}^{3}/_{4}$, because ¬EQ(dec(2),0). Similarly, the second could be instantiated by $1 - {}^{3}/_{4}$, because EQ(dec(1),0).

[Solution 1]
Input Spec: (((WN 1) ?WX) ((FR 2) ?NY / ?DY) (< ?NY ?DY))

State information for STATE1. Ruleset: CORRECT-W1&2
Instantiations: (#WN1 #WN3 #DN1 #NM2A #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*DEC ?WX) AND (*SUBTRACT ?DY ?NY) / ?DY)))
(¬EQ(DEC(?WX),0) ∧ ¬EQ(SUB(?DY,?NY),0) ∧ LT(?NY,?DY))
Exclusion clauses: NIL

State information for STATE2. Ruleset: ERROR-W1
Instantiations: NIL
Outputs: NIL

The models can be distinguished by the fact that they produce different numbers of outputs.

[Solution 2]
Input Spec: (((WN 1) ?WX) ((FR 2) ?NY / ?DY) (< ?NY ?DY))

State information for STATE1. Ruleset: CORRECT-W1&2
Instantiations: (#WN1 #WN3 #DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?DY ?NY) / ?DY)))
(¬EQ(SUB(?DY,?NY),0) ∧ LT(?NY,?DY))
Exclusion clauses: EQ(DEC(?WX),0)

State information for STATE2. Ruleset: ERROR-W1
Instantiations: NIL
Outputs: NIL

The models can be distinguished by the fact that they produce different numbers of outputs.

Error type W2

In the first solution, the critical constraint is the first one:

(¬EQ(?NY,SUB(?DY,?NY)) ∨ ¬EQ(SUB(DEC(?WX),?WY),SUB(?WX,?WY))).

In fact, it is a tautology because the second disjunct will be true for whatever values we choose for ?wx and ?wy. An example CP is: $5 - {}^{2}/_{3}$; the correcto answer is $2{}^{2}/_{3}$, but the error model computes $3{}^{1}/_{3}$. The second CP description could be instantiated by $5 - 4{}^{1}/_{3}$ yielding $^{2}/_{3}$ from the correct model but $1{}^{1}/_{3}$ from the error one. The second solution path is followed when EQ(SUB(DEC(?WX),?WY),0).

[Solution 1]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 2) ?NY / ?DY) (< ?WY ?WX) (< ?NY ?DY))

State information for STATE1. Ruleset: CORRECT-W1&2

Instantiations: (#WN1 #WN4 #DN1 #NM2A #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT (*DEC ?WX) ?WY) AND (*SUBTRACT ?DY ?NY) / ?DY)))
(¬EQ(SUB(DEC(?WX),?WY),0) ∧ ¬EQ(SUB(?DY,?NY),0) ∧ LT(?WY,?WX) ∧ LT(?NY,?DY))
Exclusion clauses: NIL

State information for STATE2. Ruleset: ERROR-W2
Instantiations: (#WN1M #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?WX ?WY) AND ?NY / ?DY)))
(¬EQ(SUB(?WX,?WY),0) ∧ ¬EQ(?NY,0) ∧ LT(?WY,?WX) ∧ LT(?NY,?DY))
Exclusion clauses: NIL

The outputs ((*SUBTRACT (*DEC ?WX) ?WY) AND (*SUBTRACT ?DY ?NY) / ?DY) and ((*SUBTRACT ?WX ?WY) AND ?NY / ?DY) can be unified. Therefore, a CP could be generated provided the following constraint-set is instantiable:
((¬EQ(?NY,SUB(?DY,?NY)) ∨ ¬EQ(SUB(DEC(?WX),?WY),SUB(?WX,?WY))) ∧ ¬EQ(?NY,0) ∧ ¬EQ(SUB(?WX,?WY),0) ∧ ¬EQ(SUB(DEC(?WX),?WY),0) ∧ ¬EQ(SUB(?DY,?NY),0) ∧ LT(?WY,?WX) ∧ LT(?NY,?DY))

[Solution 2]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 2) ?NY / ?DY) (< ?WY ?WX) (< ?NY ?DY))

State information for STATE1. Ruleset: CORRECT-W1&2
Instantiations: (#WN1 #WN4 #DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?DY ?NY) / ?DY)))
(¬EQ(SUB(?DY,?NY),0) ∧ LT(?WY,?WX) ∧ LT(?NY,?DY))
Exclusion clauses: EQ(SUB(DEC(?WX),?WY),0)

State information for STATE2. Ruleset: ERROR-W2
Instantiations: (#WN1M #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?WX ?WY) AND ?NY / ?DY)))
(¬EQ(SUB(?WX,?WY),0) ∧ ¬EQ(?NY,0) ∧ LT(?WY,?WX) ∧ LT(?NY,?DY))
Exclusion clauses: NIL

The outputs ((*SUBTRACT ?DY ?NY) / ?DY) and ((*SUBTRACT ?WX ?WY) AND ?NY / ?DY) cannot be unified. Therefore, a problem, satisfying the above constraint set, will discriminate between the two models.

Error type W3

In the error model, the child gives up with problems where the subtrahend is a loose whole number. The correct model has no problems. Thus, most problems of the form $N^{X}/_{Z} - M^{Y}/_{Z}$ will discriminate.

[Solution 1]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 1) ?NX / ?DX) (<= ?WY ?WX) (< ?NX ?DX))

State information for STATE1. Ruleset: CORRECT-W3
Instantiations: (#WN4 #WN2 #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?WX ?WY) AND ?NX / ?DX)))
(¬EQ(SUB(?WX,?WY),0) ∧ ¬EQ(?NX,0) ∧ LE(?WY,?WX) ∧ LT(?NX,?DX))
Exclusion clauses: NIL

State information for STATE2. Ruleset: ERROR-W3
Instantiations: NIL
Outputs: NIL

The models can be distinguished by the fact that they produce different numbers of outputs.

[Solution 2]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 1) ?NX / ?DX) (<= ?WY ?WX) (< ?NX ?DX))

State information for STATE1. Ruleset: CORRECT-W3
Instantiations: (#WN4 #WN2 #END1 #HLT)
Outputs: ((*OUTPUT (?NX / ?DX)))
(¬EQ(?NX,0) ∧ LE(?WY,?WX) ∧ LT(?NX,?DX))
Exclusion clauses: EQ(SUB(?WX,?WY),0)

State information for STATE2. Ruleset: ERROR-W3
Instantiations: NIL
Outputs: NIL

The models can be distinguished by the fact that they produce different numbers of outputs.

Error type W4

In the error model, the child cannot deal with mixed fraction problems where only the first parameter is mixed. Thus, most problems of that form will discriminate. The exception is the case where the first whole number is zero, i.e. EQ(?WX,0). This is the path described by solution 2. In fact, problems where ?WX is zero are not the norm in this domain; PG only considered this case because the Input Specification was not sufficiently restrictive. We chose to leave the Input Specification as it is because the more general the Input Specification, the more difficult it is to perform abstract interpretation (there are more possible paths to consider). This is true of many of the examples in this thesis.

[Solution 1]
Input Spec: (((WN 1) ?WX) ((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (< ?NX ?DX) (< ?NY ?DY))

State information for STATE1. Ruleset: CORRECT-W4
Instantiations: (#WN3 #DN1 #NM2A #END1 #END2 #HLT)
Outputs: ((*OUTPUT (?WX AND (*SUBTRACT ?NX ?NY) / ?DY)))
(EQ(?DX,?DY) ∧ ¬EQ(?WX,0) ∧ ¬EQ(SUB(?NX,?NY),0) ∧ LT(?NY,?NX) ∧ LT(?NX,?DY) ∧ LT(?NY,?DY))
Exclusion clauses: ¬EQ(?NX,?NY)

State information for STATE2. Ruleset: ERROR-W4
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?DY)))
(EQ(?DX,?DY) ∧ ¬EQ(SUB(?NX,?NY),0) ∧ LT(?NY,?NX) ∧ LT(?NX,?DY) ∧ LT(?NY,?DY))
Exclusion clauses: ¬EQ(?NX,?NY)

The outputs (?WX AND (*SUBTRACT ?NX ?NY) / ?DY) and ((*SUBTRACT ?NX ?NY) / ?DY) cannot be unified. Therefore, a problem, satisfying the above constraint set, will discriminate between the two models.

[Solution 2]
Input Spec: (((WN 1) ?WX) ((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (< ?NX ?DX) (< ?NY ?DY))

State information for STATE1. Ruleset: CORRECT-W4
Instantiations: (#WN3 #DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?DY)))
(EQ(?DX,?DY) ∧ ¬EQ(SUB(?NX,?NY),0) ∧ LT(?NY,?NX) ∧ LT(?NX,?DY) ∧ LT(?NY,?DY))
Exclusion clauses: (EQ(?WX,0) ∧ ¬EQ(?NX,?NY))

State information for STATE2. Ruleset: ERROR-W4
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?DY)))
(EQ(?DX,?DY) ∧ ¬EQ(SUB(?NX,?NY),0) ∧ LT(?NY,?NX) ∧ LT(?NX,?DY) ∧ LT(?NY,?DY))
Exclusion clauses: ¬EQ(?NX,?NY)

No CP can be generated for the outputs: ((*SUBTRACT ?NX ?NY) / ?DY) and ((*SUBTRACT ?NX ?NY) / ?DY), as they are equal.

Error type W5

In the error model, if after some other mistake the 1st whole number is less than the 2nd, then take the absolute difference. The rule PG1 is included to force this error. No CP can be generated if the two whole numbers are equal (see solution 2), i.e. **EQ(SUB(?wx,?wy),0)**. The key constraint which describes CPs for this pair of models is: **¬EQ(ABS-SUB(?wx,ADD1(?wx)),0)** (solution 1).

[Solution 1]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (<= ?WY ?WX) (< ?NX ?DX) (< ?NY ?DY))

State information for STATE1. Ruleset: CORRECT-W5
Instantiations: (#WN4 #DN1 #NM2A #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?WX ?WY) AND (*SUBTRACT ?NX ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(SUB(?wx,?wy),0) ∧ ¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LE(?wy,?wx) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy))
Exclusion clauses: NIL

State information for STATE2. Ruleset: ERROR-W5
Instantiations: (#PG1 #WN4M #DN1 #NM2A #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*ABS-SUBTRACT ?WX (*ADD1 ?WX)) AND (*SUBTRACT ?NX ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(ABS-SUB(?wx,ADD1(?wx)),0) ∧ ¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?wy,?wx) ∧ LE(?wy,?wx) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy))
Exclusion clauses: NIL

The outputs ((*SUBTRACT ?WX ?WY) AND (*SUBTRACT ?NX ?NY) / ?DY) and ((*ABS-SUBTRACT ?WX (*ADD1 ?WX)) AND (*SUBTRACT ?NX ?NY) / ?DY) can be unified. Therefore, a CP could be generated provided the following constraint-set is instantiable: (¬EQ(SUB(?wx,?wy),ABS-SUB(?wx,ADD1(?wx))) ∧ LT(?wy,?wx) ∧ ¬EQ(ABS-SUB(?wx,ADD1(?wx)),0) ∧ EQ(?dx,?dy) ∧ ¬EQ(SUB(?wx,?wy),0) ∧ ¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LE(?wy,?wx) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy))

[Solution 2]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (<= ?WY ?WX) (< ?NX ?DX) (< ?NY ?DY))

State information for STATE1. Ruleset: CORRECT-W5
Instantiations: (#WN4 #DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LE(?wy,?wx) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy))
Exclusion clauses: EQ(SUB(?wx,?wy),0)

State information for STATE2. Ruleset: ERROR-W5
Instantiations: (#WN4M #DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LE(?wy,?wx) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy))

Exclusion clauses: (EQ(ABS-SUB(?wx,?wy),0) ∧ ¬LT(?wy,?wx))

No CP can be generated for the outputs: ((*SUBTRACT ?NX ?NY) / ?DY) and ((*SUBTRACT ?NX ?NY) / ?DY), as they are equal.

Error type N1

In the error model, if the problem requires a borrow, then get stuck. The correct model covers two cases, one where there is a whole number in the answer, ¬EQ(DEC(SUB(?wx,?wy)),0), and one where there isn't (solution 2).

[Solution 1]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (< ?WY ?WX) (< ?NX ?NY) (< ?NX ?DX) (< ?NY ?DY))

State information for STATE1. Ruleset: CORRECT-N1&2&5
Instantiations: (#WN4 #DN1 #NM1 #BRW #NM2A #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*DEC (*SUBTRACT ?WX ?WY)) AND (*SUBTRACT (*ADD ?DY ?NX) ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(DEC(SUB(?wx,?wy)),0) ∧ ¬EQ(SUB(ADD(?dy,?nx),?ny),0) ∧ LT(?ny,ADD(?dy,?nx)) ∧ ¬EQ(SUB(?wx,?wy),0) ∧ LT(?wy,?wx) ∧ LT(?nx,?ny) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy))
Exclusion clauses: NIL

State information for STATE2. Ruleset: ERROR-N1
Instantiations: (#WN4 #DN1)
Outputs: NIL

The models can be distinguished by the fact that they produce different numbers of outputs.

[Solution 2]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (< ?WY ?WX) (< ?NX ?NY) (< ?NX ?DX) (< ?NY ?DY))

State information for STATE1. Ruleset: CORRECT-N1&2&5
Instantiations: (#WN4 #DN1 #NM1 #BRW #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT (*ADD ?DY ?NX) ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(SUB(ADD(?dy,?nx),?ny),0) ∧ LT(?ny,ADD(?dy,?nx)) ∧ ¬EQ(SUB(?wx,?wy),0) ∧ LT(?wy,?wx) ∧ LT(?nx,?ny) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy))
Exclusion clauses: EQ(DEC(SUB(?wx,?wy)),0)

State information for STATE2. Ruleset: ERROR-N1
Instantiations: (#WN4 #DN1)
Outputs: NIL

The models can be distinguished by the fact that they produce different numbers of outputs.

Error type N2

In the error model, the child takes the absolute difference between the two numerators. There are two cases, one where there is a whole number in the answer and one where there isn't. The key constraint is: **¬EQ(SUB(ADD(?dy,?nx),?ny),ABS-SUB(?nx,?ny))**.

[Solution 1]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (< ?WY ?WX) (< ?NX ?NY) (= ?DX ?DY) (< ?NX ?DX) (< ?NY ?DY))

State information for STATE1. Ruleset: CORRECT-N1&2&5
Instantiations: (#WN4 #DN1 #NM1 #BRW #NM2A #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*DEC (*SUBTRACT ?WX ?WY)) AND (*SUBTRACT (*ADD ?DY ?NX) ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(DEC(SUB(?wx,?wy)),0) ∧ ¬EQ(SUB(ADD(?dy,?nx),?ny),0) ∧ LT(?ny,ADD(?dy,?nx)) ∧ ¬EQ(SUB(?wx,?wy),0) ∧ LT(?wy,?wx) ∧ LT(?nx,?ny) ∧ EQ(?dy,?dy) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy))
Exclusion clauses: NIL

State information for STATE2. Ruleset: ERROR-N2
Instantiations: (#WN4 #DN1 #NM2M1 #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?WX ?WY) AND (*ABS-SUBTRACT ?NX ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(SUB(?wx,?wy),0) ∧ ¬EQ(ABS-SUB(?nx,?ny),0) ∧ LT(?wy,?wx) ∧ LT(?nx,?ny) ∧ EQ(?dy,?dy) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy))
Exclusion clauses: NIL

The outputs ((*DEC (*SUBTRACT ?WX ?WY)) AND (*SUBTRACT (*ADD ?DY ?NX) ?NY) / ?DY) and ((*SUBTRACT ?WX ?WY) AND (*ABS-SUBTRACT ?NX ?NY) / ?DY) can be unified. Therefore, a CP could be generated provided the following constraint-set is instantiable: ((¬EQ(SUB(ADD(?dy,?nx),?ny),ABS-SUB(?nx,?ny)) V ¬EQ(DEC(SUB(?wx,?wy)),SUB(?wx,?wy))) ∧ ¬EQ(ABS-SUB(?nx,?ny),0) ∧ EQ(?dx,?dy) ∧ ¬EQ(DEC(SUB(?wx,?wy)),0) ∧ ¬EQ(SUB(ADD(?dy,?nx),?ny),0) ∧ LT(?ny,ADD(?dy,?nx)) ∧ ¬EQ(SUB(?wx,?wy),0) ∧ LT(?wy,?wx) ∧ LT(?nx,?ny) ∧ EQ(?dy,?dy) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy))

[Solution 2]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (< ?WY ?WX) (< ?NX ?NY) (= ?DX ?DY) (< ?NX ?DX) (< ?NY ?DY))

State information for STATE1. Ruleset: CORRECT-N1&2&5

Instantiations: (#WN4 #DN1 #NM1 #BRW #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT (*ADD ?DY ?NX) ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(SUB(ADD(?dy,?nx),?ny),0) ∧ LT(?ny,ADD(?dy,?nx)) ∧ ¬EQ(SUB(?wx,?wy),0) ∧ LT(?wy,?wx) ∧ LT(?nx,?ny) ∧ EQ(?dy,?dy) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy))
Exclusion clauses: EQ(DEC(SUB(?wx,?wy)),0)

State information for STATE2. Ruleset: ERROR-N2
Instantiations: (#WN4 #DN1 #NM2M1 #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?WX ?WY) AND (*ABS-SUBTRACT ?NX ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(SUB(?wx,?wy),0) ∧ ¬EQ(ABS-SUB(?nx,?ny),0) ∧ LT(?wy,?wx) ∧ LT(?nx,?ny) ∧ EQ(?dy,?dy) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy))
Exclusion clauses: NIL

The outputs ((*SUBTRACT (*ADD ?DY ?NX) ?NY) / ?DY) and ((*SUBTRACT ?WX ?WY) AND (*ABS-SUBTRACT ?NX ?NY) / ?DY) cannot be unified. Therefore, a problem, satisfying the above constraint set, will discriminate between the two models.

Error type N3

In the error model, the child takes the absolute difference between the two numerators before dealing with the denominators. If the denominators are not equal then the child takes their absolute difference. This leads to quite a complex CP description, however the key feature is that either: (i) the LCM should not be equal to the absolute difference of the two denominators, or (ii) the difference between the factorised numerators should not be equal to the absolute difference between the original numerators, i.e.

¬EQ(FINDLCM(?dx,?dy),ABS-SUB(?dx,?dy))
∨ ¬EQ(SUB(MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx),MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny)),
ABS-SUB(?nx,?ny))

[Solution 1]
Input Spec: (((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (< ?NX ?NY) (< ?DX ?DY) (< ?NX ?DX) (< ?NY ?DY) (< (*DIV ?NY ?DY) (*DIV ?NX ?DX)))

State information for STATE1. Ruleset: CORRECT-N3&4-D1-F1&2
Instantiations: (#DN2 #LCM #FCT1 #FCT3 #FCT2 #FCT3 #FCT4 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DX) ?NX) (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DY) ?NY)) / (*FINDLCM ?DX ?DY))))
(¬EQ(SUB(MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx),MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny)),0) ∧ LT(MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny),MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx)) ∧ ¬EQ(2,1) ∧ ¬EQ(?dx,?dy) ∧ LT(?nx,?ny) ∧ LT(?dx,?dy) ∧ LT(?nx,?dx) ∧ LT(?ny,?dy) ∧ LT(DIV(?ny,?dy),DIV(?nx,?dx)))
Exclusion clauses: NIL

State information for STATE2. Ruleset: ERROR-N3&D2
Instantiations: (#NM2M2 #DN2M1 #END1 #HLT)
Outputs: ((*OUTPUT ((*ABS-SUBTRACT ?NX ?NY) / (*ABS-SUBTRACT ?DX ?DY))))
(¬EQ(ABS-SUB(?nx,?ny),0) ∧ ¬EQ(?dx,?dy) ∧ LT(?nx,?ny) ∧ LT(?dx,?dy) ∧ LT(?nx,?dx) ∧ LT(?ny,?dy) ∧ LT(DIV(?ny,?dy),DIV(?nx,?dx)))
Exclusion clauses: NIL

The outputs ((*SUBTRACT (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DX) ?NX) (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DY) ?NY)) / (*FINDLCM ?DX ?DY)) and ((*ABS-SUBTRACT ?NX ?NY) / (*ABS-SUBTRACT ?DX ?DY)) can be unified. Therefore, a CP could be generated provided the following constraint-set is instantiable:
((¬EQ(FINDLCM(?dx,?dy),ABS-SUB(?dx,?dy)) ∨
¬EQ(SUB(MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx),MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny)),ABS-SUB(?nx,?ny))) ∧
¬EQ(ABS-SUB(?nx,?ny),0) ∧
¬EQ(SUB(MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx),MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny)),0) ∧ LT(MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny),MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx)) ∧ ¬EQ(2,1) ∧ ¬EQ(?dx,?dy) ∧ LT(?nx,?ny) ∧ LT(?dx,?dy) ∧ LT(?nx,?dx) ∧ LT(?ny,?dy) ∧ LT(DIV(?ny,?dy),DIV(?nx,?dx)))

Error type N4

In the error model, the child subtracts the numerators and then the denominators. Thus, the CP description is very similar to that of error type N3. The difference is the presence of **SUB** in the place of **ABS-SUB**.

[Solution 1]
Input Spec: (((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (< ?NY ?NX) (< ?DY ?DX) (< ?NX ?DX) (< ?NY ?DY) (< (*DIV ?NY ?DY) (*DIV ?NX ?DX)))

State information for STATE1. Ruleset: CORRECT-N3&4-D1-F1&2
Instantiations: (#DN2 #LCM #FCT1 #FCT3 #FCT2 #FCT3 #FCT4 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DX) ?NX) (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DY) ?NY)) / (*FINDLCM ?DX ?DY))))
(¬EQ(SUB(MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx),MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny)),0) ∧ LT(MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny),MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx)) ∧ ¬EQ(2,1) ∧ ¬EQ(?dx,?dy) ∧ LT(?ny,?nx) ∧ LT(?dy,?dx) ∧ LT(?nx,?dx) ∧ LT(?ny,?dy) ∧ LT(DIV(?ny,?dy),DIV(?nx,?dx)))
Exclusion clauses: NIL

State information for STATE2. Ruleset: ERROR-N4&D3
Instantiations: (#NM2M3A #DN2M2 #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / (*SUBTRACT ?DX ?DY))))
(¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?dy,?dx) ∧ LT(?nx,?dx) ∧ LT(?ny,?dy) ∧ LT(DIV(?ny,?dy),DIV(?nx,?dx)))

Exclusion clauses: NIL

The outputs ((*SUBTRACT (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DX) ?NX) (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DY) ?NY)) / (*FINDLCM ?DX ?DY)) and ((*SUBTRACT ?NX ?NY) / (*SUBTRACT ?DX ?DY)) can be unified. Therefore, a CP could be generated provided the following constraint-set is instantiable:
((¬EQ(FINDLCM(?dx,?dy),SUB(?dx,?dy)) ∨
¬EQ(SUB(MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx),MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny)),SUB(?nx,?ny))) ∧
¬EQ(SUB(?nx,?ny),0) ∧ ¬EQ(SUB(MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx),MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny)),0) ∧
LT(MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny),MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx)) ∧ ¬EQ(2,1) ∧ ¬EQ(?dx,?dy) ∧
LT(?ny,?nx) ∧ LT(?dy,?dx) ∧ LT(?nx,?dx) ∧ LT(?ny,?dy) ∧ LT(DIV(?ny,?dy),DIV(?nx,?dx)))

Error type N5

In the error model, when borrowing, the child adds ten to the numerator. Therefore CPs will have the following characteristic: **¬EQ(SUB(ADD(?dy,?nx),?ny),SUB(ADD(10,?nx),?ny)).**

[Solution 1]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (< ?WY ?WX) (< ?NX ?NY) (= ?DX ?DY) (< ?NX ?DX) (< ?NY ?DY) (< ?NX 10) (< ?NY 10))

State information for STATE1. Ruleset: CORRECT-N1&2&5
Instantiations: (#WN4 #DN1 #NM1 #BRW #NM2A #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*DEC (*SUBTRACT ?WX ?WY)) AND (*SUBTRACT (*ADD ?DY ?NX) ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(DEC(SUB(?wx,?wy)),0) ∧ ¬EQ(SUB(ADD(?dy,?nx),?ny),0) ∧ LT(?ny,ADD(?dy,?nx)) ∧
¬EQ(SUB(?wx,?wy),0) ∧ LT(?wy,?wx) ∧ LT(?nx,?ny) ∧ EQ(?dy,?dy) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy) ∧ LT(?nx,10) ∧
LT(?ny,10))
Exclusion clauses: NIL

State information for STATE2. Ruleset: ERROR-N5
Instantiations: (#WN4 #DN1 #NM1 #BRWM #NM2A #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*DEC (*SUBTRACT ?WX ?WY)) AND (*SUBTRACT (*ADD 10 ?NX) ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(DEC(SUB(?wx,?wy)),0) ∧ ¬EQ(SUB(ADD(10,?nx),?ny),0) ∧ LT(?ny,ADD(10,?nx)) ∧
¬EQ(SUB(?wx,?wy),0) ∧ LT(?wy,?wx) ∧ LT(?nx,?ny) ∧ EQ(?dy,?dy) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy) ∧ LT(?nx,10) ∧
LT(?ny,10))
Exclusion clauses: NIL

The outputs ((*DEC (*SUBTRACT ?WX ?WY)) AND (*SUBTRACT (*ADD ?DY ?NX) ?NY) / ?DY) and ((*DEC (*SUBTRACT ?WX ?WY)) AND (*SUBTRACT (*ADD 10 ?NX) ?NY) / ?DY) can be unified. Therefore, a CP could be generated provided the following constraint-set is instantiable: (¬EQ(SUB(ADD(?dy,?nx),?ny),SUB(ADD(10,?nx),?ny)) ∧
LT(?ny,ADD(10,?nx)) ∧ ¬EQ(SUB(ADD(10,?nx),?ny),0) ∧ EQ(?dx,?dy) ∧ ¬EQ(DEC(SUB(?wx,?wy)),0) ∧
¬EQ(SUB(ADD(?dy,?nx),?ny),0) ∧ LT(?ny,ADD(?dy,?nx)) ∧ ¬EQ(SUB(?wx,?wy),0) ∧ LT(?wy,?wx) ∧ LT(?nx,?ny) ∧
EQ(?dy,?dy) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy) ∧ LT(?nx,10) ∧ LT(?ny,10))

[Solution 2]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (< ?WY ?WX) (< ?NX ?NY) (= ?DX ?DY) (< ?NX ?DX) (< ?NY ?DY) (< ?NX 10) (< ?NY 10))

State information for STATE1. Ruleset: CORRECT-N1&2&5
Instantiations: (#WN4 #DN1 #NM1 #BRW #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT (*ADD ?DY ?NX) ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(SUB(ADD(?dy,?nx),?ny),0) ∧ LT(?ny,ADD(?dy,?nx)) ∧ ¬EQ(SUB(?wx,?wy),0) ∧ LT(?wy,?wx) ∧
LT(?nx,?ny) ∧ EQ(?dy,?dy) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy) ∧ LT(?nx,10) ∧ LT(?ny,10))
Exclusion clauses: EQ(DEC(SUB(?wx,?wy)),0)

State information for STATE2. Ruleset: ERROR-N5
Instantiations: (#WN4 #DN1 #NM1 #BRWM #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT (*ADD 10 ?NX) ?NY) / ?DY)))
(EQ(?dx,?dy) ∧ ¬EQ(SUB(ADD(10,?nx),?ny),0) ∧ LT(?ny,ADD(10,?nx)) ∧ ¬EQ(SUB(?wx,?wy),0) ∧ LT(?wy,?wx) ∧
LT(?nx,?ny) ∧ EQ(?dy,?dy) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy) ∧ LT(?nx,10) ∧ LT(?ny,10))
Exclusion clauses: EQ(DEC(SUB(?wx,?wy)),0)

The outputs ((*SUBTRACT (*ADD ?DY ?NX) ?NY) / ?DY) and ((*SUBTRACT (*ADD 10 ?NX) ?NY) / ?DY) can be unified. Therefore, a CP could be generated provided the following constraint-set is instantiable:
(¬EQ(SUB(ADD(?dy,?nx),?ny),SUB(ADD(10,?nx),?ny)) ∧ LT(?ny,ADD(10,?nx)) ∧ ¬EQ(SUB(ADD(10,?nx),?ny),0) ∧
EQ(DEC(SUB(?wx,?wy)),0) ∧ EQ(?dx,?dy) ∧ ¬EQ(SUB(ADD(?dy,?nx),?ny),0) ∧ LT(?ny,ADD(?dy,?nx)) ∧
¬EQ(SUB(?wx,?wy),0) ∧ LT(?wy,?wx) ∧ LT(?nx,?ny) ∧ EQ(?dy,?dy) ∧ LT(?nx,?dy) ∧ LT(?ny,?dy) ∧ LT(?nx,10) ∧
LT(?ny,10))

Error type D1

In the error model, the child cannot calculate the LCM. Therefore, any problem which requires an LCM will be critical. This is embodied in the constraint **¬EQ(?dx,?dy)**, which in the instance was derived from the Input Specification.

[Solution 1]
Input Spec: (((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (¬ (= ?DX ?DY)) (< ?NX ?DX) (< ?NY ?DY) (< (*DIV ?NY ?DY) (*DIV ?NX ?DX)))

State information for STATE1. Ruleset: CORRECT-N3&4-D1-F1&2
Instantiations: (#DN2 #LCM #FCT1 #FCT3 #FCT2 #FCT3 #FCT4 #NM2A #END1 #HLT)

Outputs: ((*OUTPUT ((*SUBTRACT (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DX) ?NX) (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DY) ?NY)) / (*FINDLCM ?DX ?DY))))
(¬EQ(SUB(MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx),MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny)),0) ∧ LT(MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny),MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx)) ∧ ¬EQ(2,1) ∧ ¬EQ(?dx,?dy) ∧ LT(?nx,?dx) ∧ LT(?ny,?dy) ∧ LT(DIV(?ny,?dy),DIV(?nx,?dx)))
Exclusion clauses: NIL

State information for STATE2. Ruleset: ERROR-D1
Instantiations: NIL
Outputs: NIL

The models can be distinguished by the fact that they produce different numbers of outputs.

Error type F1

In the error model, the child works out the LCM but gets stuck because s/he cannot enlarge the numerators. Thus, any problem where ¬EQ(?dx,?dy) will be discriminatory.

[Solution 1]
Input Spec: (((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (¬ (= ?DX ?DY)) (< ?NX ?DX) (< ?NY ?DY) (< (*DIV ?NY ?DY) (*DIV ?NX ?DX)))

State information for STATE1. Ruleset: CORRECT-N3&4-D1-F1&2
Instantiations: (#DN2 #LCM #FCT1 #FCT3 #FCT2 #FCT3 #FCT4 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DX) ?NX) (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DY) ?NY)) / (*FINDLCM ?DX ?DY))))
(¬EQ(SUB(MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx),MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny)),0) ∧ LT(MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny),MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx)) ∧ ¬EQ(2,1) ∧ ¬EQ(?dx,?dy) ∧ LT(?nx,?dx) ∧ LT(?ny,?dy) ∧ LT(DIV(?ny,?dy),DIV(?nx,?dx)))
Exclusion clauses: NIL

State information for STATE2. Ruleset: ERROR-F1
Instantiations: (#DN2 #LCM)
Outputs: NIL

The models can be distinguished by the fact that they produce different numbers of outputs.

Error type F2

In the error model, work out the LCM but enlarge the numerators by cross-multiplying. Therefore, the following constraint is critical to the generation of a CP:

¬EQ(SUB(MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx),MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny)),
SUB(MULT(?dy,?nx),MULT(?dx,?ny)))

In other words, the result of subtracting the factorised numerators should not be equal to that of subtracting the cross-multiplied numerators.

[Solution 1]
Input Spec: (((FR 1) ?NX / ?DX) ((FR 2) ?NY / ?DY) (¬ (= ?DX ?DY)) (< ?NX ?DX) (< ?NY ?DY) (< (*DIV ?NY ?DY) (*DIV ?NX ?DX)))

State information for STATE1. Ruleset: CORRECT-N3&4-D1-F1&2
Instantiations: (#DN2 #LCM #FCT1 #FCT3 #FCT2 #FCT3 #FCT4 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DX) ?NX) (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DY) ?NY)) / (*FINDLCM ?DX ?DY))))
(¬EQ(SUB(MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx),MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny)),0) ∧ LT(MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny),MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx)) ∧ ¬EQ(2,1) ∧ ¬EQ(?dx,?dy) ∧ LT(?nx,?dx) ∧ LT(?ny,?dy) ∧ LT(DIV(?ny,?dy),DIV(?nx,?dx)))
Exclusion clauses: NIL

State information for STATE2. Ruleset: ERROR-F2
Instantiations: (#DN2 #LCM #FCTV1 #FCTV2 #FCTV3 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT (*MULTIPLY ?DY ?NX) (*MULTIPLY ?DX ?NY)) / (*FINDLCM ?DX ?DY))))
(¬EQ(SUB(MULT(?dy,?nx),MULT(?dx,?ny)),0) ∧ LT(MULT(?dx,?ny),MULT(?dy,?nx)) ∧ ¬EQ(?dx,?dy) ∧ LT(?nx,?dx) ∧ LT(?ny,?dy) ∧ LT(DIV(?ny,
?dy),DIV(?nx,?dx)))
Exclusion clauses: ¬EQ(MULT(?dy,?nx),MULT(?dx,?ny))

The outputs ((*SUBTRACT (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DX) ?NX) (*MULTIPLY (*DIVIDE (*FINDLCM ?DX ?DY) ?DY) ?NY)) / (*FINDLCM ?DX ?DY)) and ((*SUBTRACT (*MULTIPLY ?DY ?NX) (*MULTIPLY ?DX ?NY)) / (*FINDLCM ?DX ?DY)) can be unified. Therefore, a CP could be generated provided the following constraint-set is instantiable:
(¬EQ(SUB(MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx),MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny)),SUB(MULT(?dy,?nx),MULT(?dx,?ny))) ∧ ¬EQ(MULT(?dy,?nx),MULT(?dx,?ny)) ∧ LT(MULT(?dx,?ny),MULT(?dy,?nx)) ∧
¬EQ(SUB(MULT(?dy,?nx),MULT(?dx,?ny)),0) ∧
¬EQ(SUB(MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx),MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny)),0) ∧
LT(MULT(DIV(FINDLCM(?dx,?dy),?dy),?ny),MULT(DIV(FINDLCM(?dx,?dy),?dx),?nx)) ∧ ¬EQ(2,1) ∧ ¬EQ(?dx,?dy) ∧
LT(?nx,?dx) ∧ LT(?ny,?dy) ∧ LT(DIV(?ny,?dy),DIV(?nx,?dx)))

Error type F3

In the error model, the child factors even when the two denominators are equal; however, the denominators are not enlarged at all. Thus, the following expression describes the key feature of CPs for this model pair:
¬EQ(SUB(?nx,?ny),SUB(MULT(?d,?nx),MULT(?d,?ny))).

[Solution 1]
Input Spec: (((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D))

State information for STATE1. Ruleset: CORRECT-F3
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))
Exclusion clauses: NIL

State information for STATE2. Ruleset: ERROR-F3
Instantiations: (#DN1V #FCTV1 #FCTV2 #FCTV3 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT (*MULTIPLY ?D ?NX) (*MULTIPLY ?D ?NY)) / ?D)))
(¬EQ(SUB(MULT(?d,?nx),MULT(?d,?ny)),0) ∧ LT(MULT(?d,?ny),MULT(?d,?nx)) ∧ LT(?nx,?d) ∧ LT(?ny,?d))
Exclusion clauses: NIL

The outputs ((*SUBTRACT ?NX ?NY) / ?D) and ((*SUBTRACT (*MULTIPLY ?D ?NX) (*MULTIPLY ?D ?NY)) / ?D) can be unified. Therefore, a CP could be generated provided the following constraint-set is instantiable:
(¬EQ(SUB(?nx,?ny),SUB(MULT(?d,?nx),MULT(?d,?ny))) ∧ LT(MULT(?d,?ny),MULT(?d,?nx)) ∧
¬EQ(SUB(MULT(?d,?nx),MULT(?d,?ny)),0) ∧ ¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))

Error type H1

In the error model, the child does not cancel. This means that the CP should be one where the act of subtracting the two numerators leads to a term which requires cancelling, i.e.
NEEDS-CANCELLING-P(SUB(?nx,?ny),?d).

[Solution 1]
Input Spec: (((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D))

State information for STATE1. Ruleset: CORRECT-H1
Instantiations: (#DN1 #NM2A #HCF #END1 #HLT)
Outputs: ((*OUTPUT ((*CANCEL-NM (*SUBTRACT ?NX ?NY) ?D) / (*CANCEL-DN (*SUBTRACT ?NX ?NY) ?D))))
(¬EQ(CANCEL-NM(SUB(?nx,?ny),?d),0) ∧ ¬EQ(SUB(?nx,?ny),0) ∧ NEEDS-CANCELLING-P(SUB(?nx,?ny),?d) ∧
LT(SUB(?nx,?ny),?d) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))
Exclusion clauses: ¬EQ(?nx,?ny)

State information for STATE2. Ruleset: ERROR-H1
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))
Exclusion clauses: ¬EQ(?nx,?ny)

The outputs ((*CANCEL-NM (*SUBTRACT ?NX ?NY) ?D) / (*CANCEL-DN (*SUBTRACT ?NX ?NY) ?D)) and ((*SUBTRACT ?NX ?NY) / ?D) can be unified. Therefore, a CP could be generated provided the following constr aint-set is instantiable: ((¬EQ(?d,CANCEL-DN(SUB(?nx,?ny),?d)) ∨ ¬EQ(CANCEL-NM(SUB(?nx,?ny),?d),SUB(?nx,?ny))) ∧ ¬EQ(?nx,?ny) ∧ ¬EQ(CANCEL-NM(SUB(?nx,?ny),?d),0) ∧ ¬EQ(SUB(?nx,?ny),0) ∧ NEEDS-CANCELLING-P(SUB(?nx,?ny),?d) ∧ LT(SUB(?nx,?ny),?d) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))

[Solution 2]
Input Spec: (((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D))

State information for STATE1. Ruleset: CORRECT-H1
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))

Exclusion clauses: ((EQ(SUB(?nx,?ny),0) ∨ ¬NEEDS-CANCELLING-P(SUB(?nx,?ny),?d) ∨ ¬LT(SUB(?nx,?ny),?d)) ∧ ¬EQ(?nx,?ny))

State information for STATE2. Ruleset: ERROR-H1
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))

Exclusion clauses: ¬EQ(?nx,?ny)

No CP can be generated for the outputs: ((*SUBTRACT ?NX ?NY) / ?D) and ((*SUBTRACT ?NX ?NY) / ?D), as they are equal.

Error type E1

In the error model, the child does not change 0/D to zero. This means that the key feature of any CP is: **EQ(?nx,?ny)** (see solution 2).

[Solution 1]
Input Spec: (((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D))

State information for STATE1. Ruleset: CORRECT-E1&2&3
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))
Exclusion clauses: ¬EQ(?nx,?ny)

State information for STATE2. Ruleset: ERROR-E1
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))
Exclusion clauses: ¬EQ(?nx,?ny)

No CP can be generated for the outputs: ((*SUBTRACT ?NX ?NY) / ?D) and ((*SUBTRACT ?NX ?NY) / ?D), as they are equal.

[Solution 2]
Input Spec: (((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D))

State information for STATE1. Ruleset: CORRECT-E1&2&3
Instantiations: (#DN1 #NM2B #END3 #HLT)
Outputs: ((*OUTPUT (0)))
(EQ(?nx,?ny) ∧ LT(?ny,?d))
Exclusion clauses: ¬LT(?ny,?ny)

State information for STATE2. Ruleset: ERROR-E1
Instantiations: (#DN1 #NM2B)
Outputs: NIL

The models can be distinguished by the fact that they produce different numbers of outputs.

Error type E2

In the error model, the child changes 0/D to N/D. Therefore, **EQ(?nx,?ny)** is critical to generating a CP.

[Solution 1]
Input Spec: (((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D))

State information for STATE1. Ruleset: CORRECT-E1&2&3
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))
Exclusion clauses: ¬EQ(?nx,?ny)

State information for STATE2. Ruleset: ERROR-E2
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))
Exclusion clauses: ¬EQ(?nx,?ny)

No CP can be generated for the outputs: ((*SUBTRACT ?NX ?NY) / ?D) and ((*SUBTRACT ?NX ?NY) / ?D), as they are equal.

[Solution 2]
Input Spec: (((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D))

State information for STATE1. Ruleset: CORRECT-E1&2&3
Instantiations: (#DN1 #NM2B #END3 #HLT)
Outputs: ((*OUTPUT (0)))
(EQ(?nx,?ny) ∧ LT(?ny,?d))
Exclusion clauses: ¬LT(?ny,?ny)

State information for STATE2. Ruleset: ERROR-E2
Instantiations: (#DN1 #NM2B #END3M1 #END1 #HLT)
Outputs: ((*OUTPUT (?NY / ?D)))
(EQ(?nx,?ny) ∧ ¬EQ(?ny,0) ∧ LT(?ny,?d))
Exclusion clauses: ¬LT(?ny,?ny)

The outputs (0) and (?NY / ?D) cannot be unified. Therefore, a problem, satisfying the above constraint set, will discriminate between the two models.

Error type E3

In the error model, the child changes 0/D to D. This means that the key feature of any CP is: **¬EQ(?d,0) ∧ EQ(?nx,?ny).**

[Solution 1]
Input Spec: (((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D))

State information for STATE1. Ruleset: CORRECT-E1&2&3
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))
Exclusion clauses: ¬EQ(?nx,?ny)

State information for STATE2. Ruleset: ERROR-E3
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))
Exclusion clauses: ¬EQ(?nx,?ny)

No CP can be generated for the outputs: ((*SUBTRACT ?NX ?NY) / ?D) and ((*SUBTRACT ?NX ?NY) / ?D), as they are equal.

[Solution 2]
Input Spec: (((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D))

State information for STATE1. Ruleset: CORRECT-E1&2&3
Instantiations: (#DN1 #NM2B #END3 #HLT)
Outputs: ((*OUTPUT (0)))
(EQ(?nx,?ny) ∧ LT(?ny,?d))
Exclusion clauses: ¬LT(?ny,?ny)

State information for STATE2. Ruleset: ERROR-E3
Instantiations: (#DN1 #NM2B #END3M2 #HLT)
Outputs: ((*OUTPUT (?D)))
(EQ(?nx,?ny) ∧ LT(?ny,?d))
Exclusion clauses: ¬LT(?ny,?ny)

The outputs (0) and (?D) can be unified. Therefore, a CP could be generated provided the following constr aint-set is instantiable:
(¬EQ(?d,0) ∧ ¬LT(?ny,?ny) ∧ EQ(?nx,?ny) ∧ LT(?ny,?d))

Error type E4

In the error model, the child changes a whole number from 0 to W. This happens when the two whole numbers are equal.

[Solution 1]
Input Spec: (((WN 1) ?W) ((WN 2) ?W) ((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D))

State information for STATE1. Ruleset: CORRECT-E4&5
Instantiations: (#WN4 #DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d))
Exclusion clauses: ¬EQ(?nx,?ny)

State information for STATE2. Ruleset: ERROR-E4
Instantiations: (#WN4 #END4M1 #DN1 #NM2A #END1 #END2 #HLT)
Outputs: ((*OUTPUT (?W AND (*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(?w,0) ∧ ¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ EQ(SUB(?w,?w),0) ∧ LT(?nx,?d) ∧ LT(?ny,?d))
Exclusion clauses: ¬EQ(?nx,?ny)

The outputs ((*SUBTRACT ?NX ?NY) / ?D) and (?W AND (*SUBTRACT ?NX ?NY) / ?D) cannot be unified. Therefore, a problem, satisfying the above constraint set, will discriminate between the two models.

[Solution 2]
Input Spec: (((WN 1) ?W) ((WN 2) ?W) ((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D))

State information for STATE1. Ruleset: CORRECT-E4&5
Instantiations: (#WN4 #DN1 #NM2B #END4 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?W ?W))))
(EQ(?nx,?ny) ∧ LT(?ny,?d))
Exclusion clauses: ¬LT(?ny,?ny)

State information for STATE2. Ruleset: ERROR-E4
Instantiations: (#WN4 #END4M1 #DN1 #NM2B)
Outputs: NIL

The models can be distinguished by the fact that they produce different numbers of outputs.

Error type E5

In the error model, the child leaves the whole number out of the answer. Therefore, the key CP feature is: **¬EQ(SUB(?wx,?wy),0).**

[Solution 1]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D) (< ?WY ?WX))

State information for STATE1. Ruleset: CORRECT-E4&5
Instantiations: (#WN4 #DN1 #NM2A #END1 #END2 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?WX ?WY) AND (*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(SUB(?wx,?wy),0) ∧ ¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d) ∧ LT(?wy,?wx))
Exclusion clauses: ¬EQ(?nx,?ny)

State information for STATE2. Ruleset: ERROR-E5
Instantiations: (#WN4 #DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX ?NY) / ?D)))
(¬EQ(SUB(?nx,?ny),0) ∧ LT(?ny,?nx) ∧ LT(?nx,?d) ∧ LT(?ny,?d) ∧ LT(?wy,?wx))
Exclusion clauses: ¬EQ(?nx,?ny)

The outputs ((*SUBTRACT ?WX ?WY) AND (*SUBTRACT ?NX ?NY) / ?D) and ((*SUBTRACT ?NX ?NY) / ?D) cannot be unified. Therefore, a problem, satisfying the above constraint set, will discriminate between the two models.

[Solution 2]
Input Spec: (((WN 1) ?WX) ((WN 2) ?WY) ((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D) (< ?WY ?WX))

State information for STATE1. Ruleset: CORRECT-E4&5
Instantiations: (#WN4 #DN1 #NM2B #END4 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?WX ?WY))))
(EQ(?nx,?ny) ∧ LT(?ny,?d) ∧ LT(?wy,?wx))
Exclusion clauses: ¬LT(?ny,?ny)

State information for STATE2. Ruleset: ERROR-E5
Instantiations: (#WN4 #DN1 #NM2B #END4 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?WX ?WY))))
(EQ(?nx,?ny) ∧ LT(?ny,?d) ∧ LT(?wy,?wx))
Exclusion clauses: ¬LT(?ny,?ny)

No CP can be generated for the outputs: ((*SUBTRACT ?WX ?WY)) and ((*SUBTRACT ?WX ?WY)), as they are equal.

Appendix IVd

Full Trace of PG's Evaluation

This appendix lists the CPs for each model pair used to evaluate PG. Each CP is followed by the output from each of the two models in the pair. The degree of correspondence between PG and PGPS is listed after all of the CPs for a given model pair.

PG predictions for models: CORRECT-W1&2 vs. ERROR-W1...
Input: (((WN 1) 5) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (4 AND 1 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 4) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (3 AND 1 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 3) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 2) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 1) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (1 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 5) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (4 AND 2 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 4) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (3 AND 2 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 3) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (2 AND 2 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 2) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 2 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 1) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (2 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 5) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (4 AND 3 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 4) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (3 AND 3 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 3) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (2 AND 3 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 2) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 3 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 1) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (3 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 5) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (4 AND 4 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 4) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (3 AND 4 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 3) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (2 AND 4 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 2) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 4 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 1) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (4 / 5)))
ERROR-W1: (NIL)
Input: (((WN 1) 5) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (4 AND 1 / 4)))
ERROR-W1: (NIL)
Input: (((WN 1) 4) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (3 AND 1 / 4)))
ERROR-W1: (NIL)
Input: (((WN 1) 3) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (2 AND 1 / 4)))
ERROR-W1: (NIL)
Input: (((WN 1) 2) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 4)))
ERROR-W1: (NIL)
Input: (((WN 1) 1) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (1 / 4)))
ERROR-W1: (NIL)
Input: (((WN 1) 5) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (4 AND 3 / 4)))
ERROR-W1: (NIL)
Input: (((WN 1) 4) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (3 AND 3 / 4)))
ERROR-W1: (NIL)
Input: (((WN 1) 3) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (2 AND 3 / 4)))
ERROR-W1: (NIL)
Input: (((WN 1) 2) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (1 AND 3 / 4)))
ERROR-W1: (NIL)
Input: (((WN 1) 1) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (3 / 4)))
ERROR-W1: (NIL)
Input: (((WN 1) 5) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (4 AND 1 / 3)))
ERROR-W1: (NIL)
Input: (((WN 1) 4) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (3 AND 1 / 3)))
ERROR-W1: (NIL)
Input: (((WN 1) 3) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (2 AND 1 / 3)))
ERROR-W1: (NIL)
Input: (((WN 1) 2) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 3)))
ERROR-W1: (NIL)
Input: (((WN 1) 1) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (1 / 3)))
ERROR-W1: (NIL)
Input: (((WN 1) 5) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (4 AND 2 / 3)))
ERROR-W1: (NIL)
Input: (((WN 1) 4) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (3 AND 2 / 3)))
ERROR-W1: (NIL)
Input: (((WN 1) 3) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (2 AND 2 / 3)))
ERROR-W1: (NIL)
Input: (((WN 1) 2) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (1 AND 2 / 3)))
ERROR-W1: (NIL)
Input: (((WN 1) 1) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (2 / 3)))
ERROR-W1: (NIL)
Input: (((WN 1) 5) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (4 AND 1 / 2)))
ERROR-W1: (NIL)
Input: (((WN 1) 4) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (3 AND 1 / 2)))
ERROR-W1: (NIL)
Input: (((WN 1) 3) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (2 AND 1 / 2)))
ERROR-W1: (NIL)
Input: (((WN 1) 2) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 2)))
ERROR-W1: (NIL)
Input: (((WN 1) 1) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (1 / 2)))
ERROR-W1: (NIL)
The prediction matches the empirical data.
Number of Possible Problems: 45.
Number of Empirically-derived CPs: 45.
Number of PG-predicted CPs: 45.

PG predictions for models: CORRECT-W1&2 vs. ERROR-W2...
Input: (((WN 1) 5) ((WN 2) 4) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (1 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 4 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 5)))
ERROR-W2: ((*OUTPUT (2 AND 4 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (1 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 4 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W2: ((*OUTPUT (3 AND 4 / 5)))

Input: (((WN 1) 4) ((WN 2) 2) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 5)))
ERROR-W2: ((*OUTPUT (2 AND 4 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (1 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 4 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (3 AND 1 / 5)))
ERROR-W2: ((*OUTPUT (4 AND 4 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W2: ((*OUTPUT (3 AND 4 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 5)))
ERROR-W2: ((*OUTPUT (2 AND 4 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 2) 4 / 5))
CORRECT-W1&2: ((*OUTPUT (1 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 4 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (2 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 2 / 5)))
ERROR-W2: ((*OUTPUT (2 AND 3 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (2 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (2 AND 2 / 5)))
ERROR-W2: ((*OUTPUT (3 AND 3 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 2 / 5)))
ERROR-W2: ((*OUTPUT (2 AND 3 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (2 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (3 AND 2 / 5)))
ERROR-W2: ((*OUTPUT (4 AND 3 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (2 AND 2 / 5)))
ERROR-W2: ((*OUTPUT (3 AND 3 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 2 / 5)))
ERROR-W2: ((*OUTPUT (2 AND 3 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 2) 3 / 5))
CORRECT-W1&2: ((*OUTPUT (2 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (3 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 3 / 5)))
ERROR-W2: ((*OUTPUT (2 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (3 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (2 AND 3 / 5)))
ERROR-W2: ((*OUTPUT (3 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 3 / 5)))
ERROR-W2: ((*OUTPUT (2 AND 2 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (3 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (3 AND 3 / 5)))
ERROR-W2: ((*OUTPUT (4 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (2 AND 3 / 5)))
ERROR-W2: ((*OUTPUT (3 AND 2 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 3 / 5)))
ERROR-W2: ((*OUTPUT (2 AND 2 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 2) 2 / 5))
CORRECT-W1&2: ((*OUTPUT (3 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (4 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 4 / 5)))
ERROR-W2: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (4 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (2 AND 4 / 5)))
ERROR-W2: ((*OUTPUT (3 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 4 / 5)))
ERROR-W2: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (4 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (3 AND 4 / 5)))
ERROR-W2: ((*OUTPUT (4 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (2 AND 4 / 5)))
ERROR-W2: ((*OUTPUT (3 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (1 AND 4 / 5)))
ERROR-W2: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 2) 1 / 5))
CORRECT-W1&2: ((*OUTPUT (4 / 5)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (1 / 4)))
ERROR-W2: ((*OUTPUT (1 AND 3 / 4)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 4)))
ERROR-W2: ((*OUTPUT (2 AND 3 / 4)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (1 / 4)))
ERROR-W2: ((*OUTPUT (1 AND 3 / 4)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (2 AND 1 / 4)))
ERROR-W2: ((*OUTPUT (3 AND 3 / 4)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 4)))
ERROR-W2: ((*OUTPUT (2 AND 3 / 4)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (1 / 4)))
ERROR-W2: ((*OUTPUT (1 AND 3 / 4)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (3 AND 1 / 4)))
ERROR-W2: ((*OUTPUT (4 AND 3 / 4)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (2 AND 1 / 4)))
ERROR-W2: ((*OUTPUT (3 AND 3 / 4)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 4)))
ERROR-W2: ((*OUTPUT (2 AND 3 / 4)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 2) 3 / 4))
CORRECT-W1&2: ((*OUTPUT (1 / 4)))
ERROR-W2: ((*OUTPUT (1 AND 3 / 4)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (3 / 4)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 4)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (1 AND 3 / 4)))
ERROR-W2: ((*OUTPUT (2 AND 1 / 4)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (3 / 4)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 4)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (2 AND 3 / 4)))
ERROR-W2: ((*OUTPUT (3 AND 1 / 4)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (1 AND 3 / 4)))
ERROR-W2: ((*OUTPUT (2 AND 1 / 4)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (3 / 4)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 4)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (3 AND 3 / 4)))
ERROR-W2: ((*OUTPUT (4 AND 1 / 4)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (2 AND 3 / 4)))
ERROR-W2: ((*OUTPUT (3 AND 1 / 4)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (1 AND 3 / 4)))
ERROR-W2: ((*OUTPUT (2 AND 1 / 4)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 2) 1 / 4))
CORRECT-W1&2: ((*OUTPUT (3 / 4)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 4)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (1 / 3)))
ERROR-W2: ((*OUTPUT (1 AND 2 / 3)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 3)))
ERROR-W2: ((*OUTPUT (2 AND 2 / 3)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (1 / 3)))
ERROR-W2: ((*OUTPUT (1 AND 2 / 3)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (2 AND 1 / 3)))
ERROR-W2: ((*OUTPUT (3 AND 2 / 3)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 3)))
ERROR-W2: ((*OUTPUT (2 AND 2 / 3)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (1 / 3)))
ERROR-W2: ((*OUTPUT (1 AND 2 / 3)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (3 AND 1 / 3)))
ERROR-W2: ((*OUTPUT (4 AND 2 / 3)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (2 AND 1 / 3)))
ERROR-W2: ((*OUTPUT (3 AND 2 / 3)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 3)))
ERROR-W2: ((*OUTPUT (2 AND 2 / 3)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 2) 2 / 3))
CORRECT-W1&2: ((*OUTPUT (1 / 3)))
ERROR-W2: ((*OUTPUT (1 AND 2 / 3)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (2 / 3)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 3)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (1 AND 2 / 3)))
ERROR-W2: ((*OUTPUT (2 AND 1 / 3)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (2 / 3)))

ERROR-W2: ((*OUTPUT (1 AND 1 / 3)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (2 AND 2 / 3)))
ERROR-W2: ((*OUTPUT (3 AND 1 / 3)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (1 AND 2 / 3)))
ERROR-W2: ((*OUTPUT (2 AND 1 / 3)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (2 / 3)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 3)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (3 AND 2 / 3)))
ERROR-W2: ((*OUTPUT (4 AND 1 / 3)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (2 AND 2 / 3)))
ERROR-W2: ((*OUTPUT (3 AND 1 / 3)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (1 AND 2 / 3)))
ERROR-W2: ((*OUTPUT (2 AND 1 / 3)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 2) 1 / 3))
CORRECT-W1&2: ((*OUTPUT (2 / 3)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 3)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (1 / 2)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 2)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 2)))
ERROR-W2: ((*OUTPUT (2 AND 1 / 2)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (1 / 2)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 2)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (2 AND 1 / 2)))
ERROR-W2: ((*OUTPUT (3 AND 1 / 2)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 2)))
ERROR-W2: ((*OUTPUT (2 AND 1 / 2)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (1 / 2)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 2)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (3 AND 1 / 2)))
ERROR-W2: ((*OUTPUT (4 AND 1 / 2)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (2 AND 1 / 2)))
ERROR-W2: ((*OUTPUT (3 AND 1 / 2)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (1 AND 1 / 2)))
ERROR-W2: ((*OUTPUT (2 AND 1 / 2)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 2) 1 / 2))
CORRECT-W1&2: ((*OUTPUT (1 / 2)))
ERROR-W2: ((*OUTPUT (1 AND 1 / 2)))

```
The prediction matches the empirical data.
Number of Possible Problems:          90.
Number of Empirically-derived CPs:    90.
Number of PG-predicted CPs:           90.
```

PG predictions for models: CORRECT-W3 vs. ERROR-W3...

Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (2 AND 4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (3 AND 4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (2 AND 4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (4 AND 4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (3 AND 4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (2 AND 4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 4 / 5))
CORRECT-W3: ((*OUTPUT (4 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (2 AND 3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (3 AND 3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (2 AND 3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (4 AND 3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (3 AND 3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (2 AND 3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 3 / 5))
CORRECT-W3: ((*OUTPUT (3 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (2 AND 2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (3 AND 2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (2 AND 2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (4 AND 2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (3 AND 2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (2 AND 2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 2 / 5))
CORRECT-W3: ((*OUTPUT (2 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 5)))

```
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (3 AND 1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (4 AND 1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (3 AND 1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 1 / 5))
CORRECT-W3: ((*OUTPUT (1 / 5)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (1 AND 3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (2 AND 3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (1 AND 3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (3 AND 3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (2 AND 3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (1 AND 3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (4 AND 3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (3 AND 3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (2 AND 3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (1 AND 3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 3 / 4))
CORRECT-W3: ((*OUTPUT (3 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (2 AND 1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (3 AND 1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (2 AND 1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (4 AND 1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (3 AND 1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (2 AND 1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 1 / 4))
CORRECT-W3: ((*OUTPUT (1 / 4)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (1 AND 2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (2 AND 2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (1 AND 2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (3 AND 2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (2 AND 2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (1 AND 2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (4 AND 2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (3 AND 2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (2 AND 2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (1 AND 2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 2 / 3))
CORRECT-W3: ((*OUTPUT (2 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (2 AND 1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (3 AND 1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (2 AND 1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 3))
```

CORRECT-W3: ((*OUTPUT (4 AND 1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (3 AND 1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (2 AND 1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 1 / 3))
CORRECT-W3: ((*OUTPUT (1 / 3)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (2 AND 1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (3 AND 1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (2 AND 1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (4 AND 1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (3 AND 1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (2 AND 1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (1 AND 1 / 2)))
ERROR-W3: (NIL)
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 1 / 2))
CORRECT-W3: ((*OUTPUT (1 / 2)))
ERROR-W3: (NIL)
The prediction matches the empirical data.
Number of Possible Problems: 135.
Number of Empirically-derived CPs: 135.
Number of PG-predicted CPs: 135.

PG predictions for models: CORRECT-W4 vs. ERROR-W4...
Input: (((WN 1) 5) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-W4: ((*OUTPUT (5 AND 1 / 5)))
ERROR-W4: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 4) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-W4: ((*OUTPUT (4 AND 1 / 5)))
ERROR-W4: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 3) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-W4: ((*OUTPUT (3 AND 1 / 5)))
ERROR-W4: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 2) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-W4: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W4: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 1) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-W4: ((*OUTPUT (1 AND 1 / 5)))
ERROR-W4: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-W4: ((*OUTPUT (5 AND 2 / 5)))
ERROR-W4: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 4) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-W4: ((*OUTPUT (4 AND 2 / 5)))
ERROR-W4: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 3) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-W4: ((*OUTPUT (3 AND 2 / 5)))
ERROR-W4: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 2) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-W4: ((*OUTPUT (2 AND 2 / 5)))
ERROR-W4: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 1) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-W4: ((*OUTPUT (1 AND 2 / 5)))
ERROR-W4: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 5) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-W4: ((*OUTPUT (5 AND 1 / 5)))
ERROR-W4: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 4) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-W4: ((*OUTPUT (4 AND 1 / 5)))
ERROR-W4: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 3) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-W4: ((*OUTPUT (3 AND 1 / 5)))
ERROR-W4: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 2) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-W4: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W4: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 1) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-W4: ((*OUTPUT (1 AND 1 / 5)))
ERROR-W4: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-W4: ((*OUTPUT (5 AND 2 / 4)))
ERROR-W4: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 4) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-W4: ((*OUTPUT (4 AND 2 / 4)))
ERROR-W4: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 3) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-W4: ((*OUTPUT (3 AND 2 / 4)))
ERROR-W4: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 2) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-W4: ((*OUTPUT (2 AND 2 / 4)))
ERROR-W4: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 1) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-W4: ((*OUTPUT (1 AND 2 / 4)))
ERROR-W4: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 5) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-W4: ((*OUTPUT (5 AND 1 / 3)))
ERROR-W4: ((*OUTPUT (1 / 3)))
Input: (((WN 1) 4) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-W4: ((*OUTPUT (4 AND 1 / 3)))
ERROR-W4: ((*OUTPUT (1 / 3)))
Input: (((WN 1) 3) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-W4: ((*OUTPUT (3 AND 1 / 3)))
ERROR-W4: ((*OUTPUT (1 / 3)))
Input: (((WN 1) 2) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-W4: ((*OUTPUT (2 AND 1 / 3)))
ERROR-W4: ((*OUTPUT (1 / 3)))
Input: (((WN 1) 1) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-W4: ((*OUTPUT (1 AND 1 / 3)))
ERROR-W4: ((*OUTPUT (1 / 3)))
The prediction matches the empirical data.
Number of Possible Problems: 405.
Number of Empirically-derived CPs: 40.
Number of PG-predicted CPs: 40.

PG predictions for models: CORRECT-W5 vs. ERROR-W5...
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-W5: ((*OUTPUT (3 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))

Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-W5: ((*OUTPUT (4 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-W5: ((*OUTPUT (3 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-W5: ((*OUTPUT (3 AND 2 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-W5: ((*OUTPUT (4 AND 2 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-W5: ((*OUTPUT (3 AND 2 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-W5: ((*OUTPUT (3 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-W5: ((*OUTPUT (4 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-W5: ((*OUTPUT (3 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (3 AND 3 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (4 AND 3 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (3 AND 3 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (3 AND 2 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (4 AND 2 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (3 AND 2 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (3 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (4 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (3 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-W5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-W5: ((*OUTPUT (2 AND 2 / 4)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 4)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-W5: ((*OUTPUT (3 AND 2 / 4)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 4)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-W5: ((*OUTPUT (2 AND 2 / 4)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 4)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-W5: ((*OUTPUT (4 AND 2 / 4)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 4)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-W5: ((*OUTPUT (3 AND 2 / 4)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 4)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-W5: ((*OUTPUT (2 AND 2 / 4)))
ERROR-W5: ((*OUTPUT (1 AND 2 / 4)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-W5: ((*OUTPUT (2 AND 1 / 3)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 3)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-W5: ((*OUTPUT (3 AND 1 / 3)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 3)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-W5: ((*OUTPUT (2 AND 1 / 3)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 3)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-W5: ((*OUTPUT (4 AND 1 / 3)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 3)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-W5: ((*OUTPUT (3 AND 1 / 3)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 3)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-W5: ((*OUTPUT (2 AND 1 / 3)))
ERROR-W5: ((*OUTPUT (1 AND 1 / 3)))

The prediction matches the empirical data.
Number of Possible Problems: 1215.
Number of Empirically-derived CPs: 48.
Number of PG-predicted CPs: 48.

PG predictions for models: CORRECT-N1&2&5 vs. ERROR-N1...

Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 4 / 5))

CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 2 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (2 / 4)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 4)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (2 / 4)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 4)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 4)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (2 / 4)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 2 / 4)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 4)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 4)))
ERROR-N1: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (2 / 4)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (2 / 3)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 3)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (2 / 3)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 3)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 3)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (2 / 3)))
ERROR-N1: (NIL)
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 2 / 3)))
ERROR-N1: (NIL)
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 3)))
ERROR-N1: (NIL)
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 3)))
ERROR-N1: (NIL)
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (2 / 3)))
ERROR-N1: (NIL)

```
The prediction matches the empirical data.
Number of Possible Problems:          280.
Number of Empirically-derived CPs:     80.
Number of PG-predicted CPs:            80.
```

PG predictions for models: CORRECT-N1&2&5 vs. ERROR-N2..

Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (3 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 5)))

Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (4 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (3 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-N2: ((*OUTPUT (3 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 2 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 3 / 5)))
ERROR-N2: ((*OUTPUT (4 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-N2: ((*OUTPUT (3 AND 2 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 2 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 3 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-N2: ((*OUTPUT (3 AND 3 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 3 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 2 / 5)))
ERROR-N2: ((*OUTPUT (4 AND 3 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-N2: ((*OUTPUT (3 AND 3 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 3 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (3 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (4 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (3 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-N2: ((*OUTPUT (3 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 2 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 3 / 5)))
ERROR-N2: ((*OUTPUT (4 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-N2: ((*OUTPUT (3 AND 2 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 2 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (3 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (4 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (3 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N2: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 2 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (2 / 4)))
ERROR-N2: ((*OUTPUT (1 AND 2 / 4)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 4)))
ERROR-N2: ((*OUTPUT (2 AND 2 / 4)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (2 / 4)))
ERROR-N2: ((*OUTPUT (1 AND 2 / 4)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 4)))
ERROR-N2: ((*OUTPUT (3 AND 2 / 4)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 4)))
ERROR-N2: ((*OUTPUT (2 AND 2 / 4)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (2 / 4)))
ERROR-N2: ((*OUTPUT (1 AND 2 / 4)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 2 / 4)))
ERROR-N2: ((*OUTPUT (4 AND 2 / 4)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 4)))
ERROR-N2: ((*OUTPUT (3 AND 2 / 4)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 4)))
ERROR-N2: ((*OUTPUT (2 AND 2 / 4)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 4) ((FR 2) 3 / 4))
CORRECT-N1&2&5: ((*OUTPUT (2 / 4)))
ERROR-N2: ((*OUTPUT (1 AND 2 / 4)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (2 / 3)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 3)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 3)))
ERROR-N2: ((*OUTPUT (2 AND 1 / 3)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (2 / 3)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 3)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 3)))
ERROR-N2: ((*OUTPUT (3 AND 1 / 3)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 3)))

ERROR-N2: ((*OUTPUT (2 AND 1 / 3)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (2 / 3)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 3)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 2 / 3)))
ERROR-N2: ((*OUTPUT (4 AND 1 / 3)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 3)))
ERROR-N2: ((*OUTPUT (3 AND 1 / 3)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 3)))
ERROR-N2: ((*OUTPUT (2 AND 1 / 3)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 3) ((FR 2) 2 / 3))
CORRECT-N1&2&5: ((*OUTPUT (2 / 3)))
ERROR-N2: ((*OUTPUT (1 AND 1 / 3)))
The prediction matches the empirical data.
Number of Possible Problems: 80.
Number of Empirically-derived CPs: 80.
Number of PG-predicted CPs: 80.

PG predictions for models: CORRECT-N3&4-D1-F1&2 vs. ERROR-N3&D2...
Input: (((FR 1) 2 / 3) ((FR 2) 3 / 5))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 15)))
ERROR-N3&D2: ((*OUTPUT (1 / 2)))
Input: (((FR 1) 1 / 2) ((FR 2) 2 / 5))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 10)))
ERROR-N3&D2: ((*OUTPUT (1 / 3)))
The prediction matches the empirical data.
Number of Possible Problems: 2.
Number of Empirically-derived CPs: 2.
Number of PG-predicted CPs: 2.

PG predictions for models: CORRECT-N3&4-D1-F1&2 vs. ERROR-N4&D3...
Input: (((FR 1) 4 / 5) ((FR 2) 3 / 4))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 20)))
ERROR-N4&D3: ((*OUTPUT (1 / 1)))
Input: (((FR 1) 4 / 5) ((FR 2) 1 / 4))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (11 / 20)))
ERROR-N4&D3: ((*OUTPUT (3 / 1)))
Input: (((FR 1) 3 / 5) ((FR 2) 1 / 4))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (7 / 20)))
ERROR-N4&D3: ((*OUTPUT (2 / 1)))
Input: (((FR 1) 2 / 5) ((FR 2) 1 / 4))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (3 / 20)))
ERROR-N4&D3: ((*OUTPUT (1 / 1)))
Input: (((FR 1) 4 / 5) ((FR 2) 2 / 3))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (2 / 15)))
ERROR-N4&D3: ((*OUTPUT (2 / 2)))
Input: (((FR 1) 3 / 4) ((FR 2) 2 / 3))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 12)))
ERROR-N4&D3: ((*OUTPUT (1 / 1)))
Input: (((FR 1) 4 / 5) ((FR 2) 1 / 3))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (7 / 15)))
ERROR-N4&D3: ((*OUTPUT (3 / 2)))
Input: (((FR 1) 3 / 5) ((FR 2) 1 / 3))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (4 / 15)))
ERROR-N4&D3: ((*OUTPUT (2 / 2)))
Input: (((FR 1) 2 / 5) ((FR 2) 1 / 3))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 15)))
ERROR-N4&D3: ((*OUTPUT (1 / 2)))
Input: (((FR 1) 3 / 4) ((FR 2) 1 / 3))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (5 / 12)))
ERROR-N4&D3: ((*OUTPUT (2 / 1)))
Input: (((FR 1) 4 / 5) ((FR 2) 1 / 2))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (3 / 10)))
ERROR-N4&D3: ((*OUTPUT (3 / 3)))
Input: (((FR 1) 3 / 5) ((FR 2) 1 / 2))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 10)))
ERROR-N4&D3: ((*OUTPUT (2 / 3)))
Input: (((FR 1) 3 / 4) ((FR 2) 1 / 2))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 4)))
ERROR-N4&D3: ((*OUTPUT (2 / 2)))
Input: (((FR 1) 2 / 3) ((FR 2) 1 / 2))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 6)))
ERROR-N4&D3: ((*OUTPUT (1 / 1)))
The prediction matches the empirical data.
Number of Possible Problems: 14.
Number of Empirically-derived CPs: 14.
Number of PG-predicted CPs: 14.

PG predictions for models: CORRECT-N1&2&5 vs. ERROR-N5...
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N5: ((*OUTPUT (9 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N5: ((*OUTPUT (1 AND 9 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N5: ((*OUTPUT (9 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N5: ((*OUTPUT (2 AND 9 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N5: ((*OUTPUT (1 AND 9 / 5)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N5: ((*OUTPUT (9 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 4 / 5)))
ERROR-N5: ((*OUTPUT (3 AND 9 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N5: ((*OUTPUT (2 AND 9 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N5: ((*OUTPUT (1 AND 9 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N5: ((*OUTPUT (9 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N5: ((*OUTPUT (8 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N5: ((*OUTPUT (1 AND 8 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N5: ((*OUTPUT (8 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-N5: ((*OUTPUT (2 AND 8 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N5: ((*OUTPUT (1 AND 8 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N5: ((*OUTPUT (8 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 3 / 5)))
ERROR-N5: ((*OUTPUT (3 AND 8 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-N5: ((*OUTPUT (2 AND 8 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-N5: ((*OUTPUT (1 AND 8 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
ERROR-N5: ((*OUTPUT (8 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 / 5)))
ERROR-N5: ((*OUTPUT (7 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-N5: ((*OUTPUT (1 AND 7 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 / 5)))
ERROR-N5: ((*OUTPUT (7 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-N5: ((*OUTPUT (2 AND 7 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-N5: ((*OUTPUT (1 AND 7 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 / 5)))
ERROR-N5: ((*OUTPUT (7 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 2 / 5)))
ERROR-N5: ((*OUTPUT (3 AND 7 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-N5: ((*OUTPUT (2 AND 7 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-N5: ((*OUTPUT (1 AND 7 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 4 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 / 5)))
ERROR-N5: ((*OUTPUT (7 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N5: ((*OUTPUT (9 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N5: ((*OUTPUT (1 AND 9 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N5: ((*OUTPUT (9 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N5: ((*OUTPUT (2 AND 9 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
ERROR-N5: ((*OUTPUT (1 AND 9 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
ERROR-N5: ((*OUTPUT (9 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (3 AND 4 / 5)))
ERROR-N5: ((*OUTPUT (3 AND 9 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 3 / 5))
CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
ERROR-N5: ((*OUTPUT (2 AND 9 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 3 / 5))

```
  CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
  ERROR-N5: ((*OUTPUT (1 AND 9 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 2 / 5) (FR 2) 3 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
  ERROR-N5: ((*OUTPUT (9 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 5) (FR 2) 3 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
  ERROR-N5: ((*OUTPUT (8 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 5) (FR 2) 3 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
  ERROR-N5: ((*OUTPUT (1 AND 8 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 5) (FR 2) 3 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
  ERROR-N5: ((*OUTPUT (8 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 5) (FR 2) 3 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (2 AND 3 / 5)))
  ERROR-N5: ((*OUTPUT (2 AND 8 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 5) (FR 2) 3 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
  ERROR-N5: ((*OUTPUT (1 AND 8 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 5) (FR 2) 3 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
  ERROR-N5: ((*OUTPUT (8 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 5) (FR 2) 3 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (3 AND 3 / 5)))
  ERROR-N5: ((*OUTPUT (3 AND 8 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 5) (FR 2) 3 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (2 AND 3 / 5)))
  ERROR-N5: ((*OUTPUT (2 AND 8 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 5) (FR 2) 3 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (1 AND 3 / 5)))
  ERROR-N5: ((*OUTPUT (1 AND 8 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 5) (FR 2) 3 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (3 / 5)))
  ERROR-N5: ((*OUTPUT (8 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 5) (FR 2) 2 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
  ERROR-N5: ((*OUTPUT (9 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 5) (FR 2) 2 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
  ERROR-N5: ((*OUTPUT (1 AND 9 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 5) (FR 2) 2 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
  ERROR-N5: ((*OUTPUT (9 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 5) (FR 2) 2 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
  ERROR-N5: ((*OUTPUT (2 AND 9 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 5) (FR 2) 2 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
  ERROR-N5: ((*OUTPUT (1 AND 9 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 5) (FR 2) 2 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
  ERROR-N5: ((*OUTPUT (9 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 5) (FR 2) 2 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (3 AND 4 / 5)))
  ERROR-N5: ((*OUTPUT (3 AND 9 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 5) (FR 2) 2 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (2 AND 4 / 5)))
  ERROR-N5: ((*OUTPUT (2 AND 9 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 5) (FR 2) 2 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (1 AND 4 / 5)))
  ERROR-N5: ((*OUTPUT (1 AND 9 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 5) (FR 2) 2 / 5))
  CORRECT-N1&2&5: ((*OUTPUT (4 / 5)))
  ERROR-N5: ((*OUTPUT (9 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 4) (FR 2) 3 / 4))
  CORRECT-N1&2&5: ((*OUTPUT (2 / 4)))
  ERROR-N5: ((*OUTPUT (8 / 4)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 4) (FR 2) 3 / 4))
  CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 4)))
  ERROR-N5: ((*OUTPUT (1 AND 8 / 4)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 4) (FR 2) 3 / 4))
  CORRECT-N1&2&5: ((*OUTPUT (2 / 4)))
  ERROR-N5: ((*OUTPUT (8 / 4)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 4) (FR 2) 3 / 4))
  CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 4)))
  ERROR-N5: ((*OUTPUT (2 AND 8 / 4)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 4) (FR 2) 3 / 4))
  CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 4)))
  ERROR-N5: ((*OUTPUT (1 AND 8 / 4)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 4) (FR 2) 3 / 4))
  CORRECT-N1&2&5: ((*OUTPUT (2 / 4)))
  ERROR-N5: ((*OUTPUT (8 / 4)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 4) (FR 2) 3 / 4))
  CORRECT-N1&2&5: ((*OUTPUT (3 AND 2 / 4)))
  ERROR-N5: ((*OUTPUT (3 AND 8 / 4)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 4) (FR 2) 3 / 4))
  CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 4)))
  ERROR-N5: ((*OUTPUT (2 AND 8 / 4)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 4) (FR 2) 3 / 4))
  CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 4)))
  ERROR-N5: ((*OUTPUT (1 AND 8 / 4)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 4) (FR 2) 3 / 4))
  CORRECT-N1&2&5: ((*OUTPUT (2 / 4)))
  ERROR-N5: ((*OUTPUT (8 / 4)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 1 / 3) (FR 2) 2 / 3))
  CORRECT-N1&2&5: ((*OUTPUT (2 / 3)))
  ERROR-N5: ((*OUTPUT (9 / 3)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 1 / 3) (FR 2) 2 / 3))
  CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 3)))
  ERROR-N5: ((*OUTPUT (1 AND 9 / 3)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 1 / 3) (FR 2) 2 / 3))
  CORRECT-N1&2&5: ((*OUTPUT (2 / 3)))
  ERROR-N5: ((*OUTPUT (9 / 3)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 1 / 3) (FR 2) 2 / 3))
  CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 3)))
  ERROR-N5: ((*OUTPUT (2 AND 9 / 3)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 1 / 3) (FR 2) 2 / 3))
  CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 3)))
  ERROR-N5: ((*OUTPUT (1 AND 9 / 3)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 1 / 3) (FR 2) 2 / 3))
  CORRECT-N1&2&5: ((*OUTPUT (2 / 3)))
  ERROR-N5: ((*OUTPUT (9 / 3)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 1 / 3) (FR 2) 2 / 3))
  CORRECT-N1&2&5: ((*OUTPUT (3 AND 2 / 3)))
  ERROR-N5: ((*OUTPUT (3 AND 9 / 3)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 1 / 3) (FR 2) 2 / 3))
  CORRECT-N1&2&5: ((*OUTPUT (2 AND 2 / 3)))
  ERROR-N5: ((*OUTPUT (2 AND 9 / 3)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 1 / 3) (FR 2) 2 / 3))
  CORRECT-N1&2&5: ((*OUTPUT (1 AND 2 / 3)))
  ERROR-N5: ((*OUTPUT (1 AND 9 / 3)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 1 / 3) (FR 2) 2 / 3))
  CORRECT-N1&2&5: ((*OUTPUT (2 / 3)))
  ERROR-N5: ((*OUTPUT (9 / 3)))
The prediction matches the empirical data.
Number of Possible Problems:          80.
Number of Empirically-derived CPs:    80.
Number of PG-predicted CPs:           80.

PG predictions for models: CORRECT-N3&4-D1-F1&2 vs. ERROR-D1...
Input: (((FR 1) 3 / 4) ((FR 2) 3 / 5))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (3 / 20)))
  ERROR-D1: (NIL)
Input: (((FR 1) 2 / 3) ((FR 2) 3 / 5))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 15)))
  ERROR-D1: (NIL)
Input: (((FR 1) 3 / 4) ((FR 2) 2 / 5))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (7 / 20)))
  ERROR-D1: (NIL)
Input: (((FR 1) 2 / 3) ((FR 2) 2 / 5))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (4 / 15)))
  ERROR-D1: (NIL)
Input: (((FR 1) 1 / 2) ((FR 2) 2 / 5))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 10)))
  ERROR-D1: (NIL)
Input: (((FR 1) 3 / 4) ((FR 2) 1 / 5))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (11 / 20)))
  ERROR-D1: (NIL)
Input: (((FR 1) 1 / 4) ((FR 2) 1 / 5))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 20)))
  ERROR-D1: (NIL)
Input: (((FR 1) 2 / 3) ((FR 2) 1 / 5))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (7 / 15)))
  ERROR-D1: (NIL)
Input: (((FR 1) 1 / 3) ((FR 2) 1 / 5))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (2 / 15)))
  ERROR-D1: (NIL)
Input: (((FR 1) 1 / 2) ((FR 2) 1 / 5))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (3 / 10)))
  ERROR-D1: (NIL)
Input: (((FR 1) 4 / 5) ((FR 2) 3 / 4))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 20)))
  ERROR-D1: (NIL)
Input: (((FR 1) 4 / 5) ((FR 2) 1 / 4))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (11 / 20)))
  ERROR-D1: (NIL)
Input: (((FR 1) 3 / 5) ((FR 2) 1 / 4))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (7 / 20)))
  ERROR-D1: (NIL)
Input: (((FR 1) 2 / 5) ((FR 2) 1 / 4))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (3 / 20)))
  ERROR-D1: (NIL)
Input: (((FR 1) 2 / 3) ((FR 2) 1 / 4))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (5 / 12)))
  ERROR-D1: (NIL)
Input: (((FR 1) 1 / 3) ((FR 2) 1 / 4))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 12)))
  ERROR-D1: (NIL)
Input: (((FR 1) 1 / 2) ((FR 2) 1 / 4))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 4)))
  ERROR-D1: (NIL)
Input: (((FR 1) 4 / 5) ((FR 2) 2 / 3))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (2 / 15)))
  ERROR-D1: (NIL)
Input: (((FR 1) 3 / 4) ((FR 2) 2 / 3))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 12)))
  ERROR-D1: (NIL)
Input: (((FR 1) 4 / 5) ((FR 2) 1 / 3))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (7 / 15)))
  ERROR-D1: (NIL)
Input: (((FR 1) 3 / 5) ((FR 2) 1 / 3))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (4 / 15)))
  ERROR-D1: (NIL)
Input: (((FR 1) 2 / 5) ((FR 2) 1 / 3))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 15)))
  ERROR-D1: (NIL)
Input: (((FR 1) 3 / 4) ((FR 2) 1 / 3))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (5 / 12)))
  ERROR-D1: (NIL)
Input: (((FR 1) 1 / 2) ((FR 2) 1 / 3))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 6)))
  ERROR-D1: (NIL)
Input: (((FR 1) 4 / 5) ((FR 2) 1 / 2))
  CORRECT-N3&4-D1-F1&2: ((*OUTPUT (3 / 10)))
```

ERROR-D1: (NIL)
Input: (((FR 1) 3 / 5) ((FR 2) 1 / 2))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 10)))
ERROR-D1: (NIL)
Input: (((FR 1) 3 / 4) ((FR 2) 1 / 2))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 4)))
ERROR-D1: (NIL)
Input: (((FR 1) 2 / 3) ((FR 2) 1 / 2))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 6)))
ERROR-D1: (NIL)
The prediction matches the empirical data.
Number of Possible Problems: 28.
Number of Empirically-derived CPs: 28.
Number of PG-predicted CPs: 28.

PG predictions for models: CORRECT-N3&4-D1-F1&2 vs. ERROR-F1...
Input: (((FR 1) 3 / 4) ((FR 2) 3 / 5))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (3 / 20)))
ERROR-F1: (NIL)
Input: (((FR 1) 2 / 3) ((FR 2) 3 / 5))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 15)))
ERROR-F1: (NIL)
Input: (((FR 1) 3 / 4) ((FR 2) 2 / 5))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (7 / 20)))
ERROR-F1: (NIL)
Input: (((FR 1) 2 / 3) ((FR 2) 2 / 5))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (4 / 15)))
ERROR-F1: (NIL)
Input: (((FR 1) 1 / 2) ((FR 2) 2 / 5))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 10)))
ERROR-F1: (NIL)
Input: (((FR 1) 3 / 4) ((FR 2) 1 / 5))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (11 / 20)))
ERROR-F1: (NIL)
Input: (((FR 1) 1 / 4) ((FR 2) 1 / 5))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 20)))
ERROR-F1: (NIL)
Input: (((FR 1) 2 / 3) ((FR 2) 1 / 5))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (7 / 15)))
ERROR-F1: (NIL)
Input: (((FR 1) 1 / 3) ((FR 2) 1 / 5))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (2 / 15)))
ERROR-F1: (NIL)
Input: (((FR 1) 1 / 2) ((FR 2) 1 / 5))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (3 / 10)))
ERROR-F1: (NIL)
Input: (((FR 1) 4 / 5) ((FR 2) 3 / 4))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 20)))
ERROR-F1: (NIL)
Input: (((FR 1) 4 / 5) ((FR 2) 1 / 4))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (11 / 20)))
ERROR-F1: (NIL)
Input: (((FR 1) 3 / 5) ((FR 2) 1 / 4))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (7 / 20)))
ERROR-F1: (NIL)
Input: (((FR 1) 2 / 5) ((FR 2) 1 / 4))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (3 / 20)))
ERROR-F1: (NIL)
Input: (((FR 1) 2 / 3) ((FR 2) 1 / 4))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (5 / 12)))
ERROR-F1: (NIL)
Input: (((FR 1) 1 / 3) ((FR 2) 1 / 4))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 12)))
ERROR-F1: (NIL)
Input: (((FR 1) 1 / 2) ((FR 2) 1 / 4))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 4)))
ERROR-F1: (NIL)
Input: (((FR 1) 4 / 5) ((FR 2) 2 / 3))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (2 / 15)))
ERROR-F1: (NIL)
Input: (((FR 1) 3 / 4) ((FR 2) 2 / 3))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 12)))
ERROR-F1: (NIL)
Input: (((FR 1) 4 / 5) ((FR 2) 1 / 3))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (7 / 15)))
ERROR-F1: (NIL)
Input: (((FR 1) 3 / 5) ((FR 2) 1 / 3))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (4 / 15)))
ERROR-F1: (NIL)
Input: (((FR 1) 2 / 5) ((FR 2) 1 / 3))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 15)))
ERROR-F1: (NIL)
Input: (((FR 1) 3 / 4) ((FR 2) 1 / 3))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (5 / 12)))
ERROR-F1: (NIL)
Input: (((FR 1) 1 / 2) ((FR 2) 1 / 3))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 6)))
ERROR-F1: (NIL)
Input: (((FR 1) 4 / 5) ((FR 2) 1 / 2))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (3 / 10)))
ERROR-F1: (NIL)
Input: (((FR 1) 3 / 5) ((FR 2) 1 / 2))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 10)))
ERROR-F1: (NIL)
Input: (((FR 1) 3 / 4) ((FR 2) 1 / 2))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 4)))
ERROR-F1: (NIL)
Input: (((FR 1) 2 / 3) ((FR 2) 1 / 2))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 6)))
ERROR-F1: (NIL)
The prediction matches the empirical data.
Number of Possible Problems: 28.
Number of Empirically-derived CPs: 28.
Number of PG-predicted CPs: 28.

PG predictions for models: CORRECT-N3&4-D1-F1&2 vs. ERROR-F2...
Input: (((FR 1) 1 / 2) ((FR 2) 1 / 4))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 4)))
ERROR-F2: ((*OUTPUT (2 / 4)))
Input: (((FR 1) 3 / 4) ((FR 2) 1 / 2))
CORRECT-N3&4-D1-F1&2: ((*OUTPUT (1 / 4)))
ERROR-F2: ((*OUTPUT (2 / 4)))
The prediction matches the empirical data.
Number of Possible Problems: 28.
Number of Empirically-derived CPs: 2.
Number of PG-predicted CPs: 2.

PG predictions for models: CORRECT-F3 vs. ERROR-F3...
Input: (((FR 1) 4 / 5) ((FR 2) 4 / 5))
CORRECT-F3: ((*OUTPUT (0 / 5)))
ERROR-F3: ((*OUTPUT (16 / 5)))
Input: (((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-F3: ((*OUTPUT (1 / 5)))
ERROR-F3: ((*OUTPUT (5 / 5)))
Input: (((FR 1) 3 / 5) ((FR 2) 3 / 5))
CORRECT-F3: ((*OUTPUT (0 / 5)))
ERROR-F3: ((*OUTPUT (12 / 5)))
Input: (((FR 1) 3 / 4) ((FR 2) 3 / 4))
CORRECT-F3: ((*OUTPUT (0 / 4)))
ERROR-F3: ((*OUTPUT (9 / 4)))
Input: (((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-F3: ((*OUTPUT (2 / 5)))
ERROR-F3: ((*OUTPUT (10 / 5)))
Input: (((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-F3: ((*OUTPUT (1 / 5)))
ERROR-F3: ((*OUTPUT (5 / 5)))
Input: (((FR 1) 2 / 5) ((FR 2) 2 / 5))
CORRECT-F3: ((*OUTPUT (0 / 5)))
ERROR-F3: ((*OUTPUT (8 / 5)))
Input: (((FR 1) 2 / 3) ((FR 2) 2 / 3))
CORRECT-F3: ((*OUTPUT (0 / 3)))
ERROR-F3: ((*OUTPUT (4 / 3)))
Input: (((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-F3: ((*OUTPUT (3 / 5)))
ERROR-F3: ((*OUTPUT (15 / 5)))
Input: (((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-F3: ((*OUTPUT (2 / 5)))
ERROR-F3: ((*OUTPUT (10 / 5)))
Input: (((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-F3: ((*OUTPUT (1 / 5)))
ERROR-F3: ((*OUTPUT (5 / 5)))
Input: (((FR 1) 1 / 5) ((FR 2) 1 / 5))
CORRECT-F3: ((*OUTPUT (0 / 5)))
ERROR-F3: ((*OUTPUT (4 / 5)))
Input: (((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-F3: ((*OUTPUT (2 / 4)))
ERROR-F3: ((*OUTPUT (8 / 4)))
Input: (((FR 1) 1 / 4) ((FR 2) 1 / 4))
CORRECT-F3: ((*OUTPUT (0 / 4)))
ERROR-F3: ((*OUTPUT (3 / 4)))
Input: (((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-F3: ((*OUTPUT (1 / 3)))
ERROR-F3: ((*OUTPUT (3 / 3)))
Input: (((FR 1) 1 / 3) ((FR 2) 1 / 3))
CORRECT-F3: ((*OUTPUT (0 / 3)))
ERROR-F3: ((*OUTPUT (2 / 3)))
Input: (((FR 1) 1 / 2) ((FR 2) 1 / 2))
CORRECT-F3: ((*OUTPUT (0 / 2)))
ERROR-F3: ((*OUTPUT (1 / 2)))
The prediction matches the empirical data.
Number of Possible Problems: 17.
Number of Empirically-derived CPs: 17.
Number of PG-predicted CPs: 17.

PG predictions for models: CORRECT-H1 vs. ERROR-H1...
Input: (((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-H1: ((*OUTPUT (1 / 2)))
ERROR-H1: ((*OUTPUT (2 / 4)))
The prediction matches the empirical data.
Number of Possible Problems: 25.
Number of Empirically-derived CPs: 1.
Number of PG-predicted CPs: 1.

PG predictions for models: CORRECT-E1&2&3 vs. ERROR-E1...
Input: (((FR 1) 4 / 5) ((FR 2) 4 / 5))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E1: (NIL)
Input: (((FR 1) 3 / 5) ((FR 2) 3 / 5))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E1: (NIL)
Input: (((FR 1) 3 / 4) ((FR 2) 3 / 4))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E1: (NIL)
Input: (((FR 1) 2 / 5) ((FR 2) 2 / 5))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E1: (NIL)
Input: (((FR 1) 2 / 3) ((FR 2) 2 / 3))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E1: (NIL)

Input: (((FR 1) 1 / 5) ((FR 2) 1 / 5))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E1: (NIL)
Input: (((FR 1) 1 / 4) ((FR 2) 1 / 4))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E1: (NIL)
Input: (((FR 1) 1 / 3) ((FR 2) 1 / 3))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E1: (NIL)
Input: (((FR 1) 1 / 2) ((FR 2) 1 / 2))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E1: (NIL)
The prediction matches the empirical data.
Number of Possible Problems: 25.
Number of Empirically-derived CPs: 9.
Number of PG-predicted CPs: 9.

PG predictions for models: CORRECT-E1&2&3 vs. ERROR-E2...
Input: (((FR 1) 4 / 5) ((FR 2) 4 / 5))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E2: ((*OUTPUT (4 / 5)))
Input: (((FR 1) 3 / 5) ((FR 2) 3 / 5))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E2: ((*OUTPUT (3 / 5)))
Input: (((FR 1) 3 / 4) ((FR 2) 3 / 4))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E2: ((*OUTPUT (3 / 4)))
Input: (((FR 1) 2 / 5) ((FR 2) 2 / 5))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E2: ((*OUTPUT (2 / 5)))
Input: (((FR 1) 2 / 3) ((FR 2) 2 / 3))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E2: ((*OUTPUT (2 / 3)))
Input: (((FR 1) 1 / 5) ((FR 2) 1 / 5))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E2: ((*OUTPUT (1 / 5)))
Input: (((FR 1) 1 / 4) ((FR 2) 1 / 4))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E2: ((*OUTPUT (1 / 4)))
Input: (((FR 1) 1 / 3) ((FR 2) 1 / 3))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E2: ((*OUTPUT (1 / 3)))
Input: (((FR 1) 1 / 2) ((FR 2) 1 / 2))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E2: ((*OUTPUT (1 / 2)))
The prediction matches the empirical data.
Number of Possible Problems: 25.
Number of Empirically-derived CPs: 9.
Number of PG-predicted CPs: 9.

PG predictions for models: CORRECT-E1&2&3 vs. ERROR-E3...
Input: (((FR 1) 4 / 5) ((FR 2) 4 / 5))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E3: ((*OUTPUT (5)))
Input: (((FR 1) 3 / 5) ((FR 2) 3 / 5))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E3: ((*OUTPUT (5)))
Input: (((FR 1) 3 / 4) ((FR 2) 3 / 4))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E3: ((*OUTPUT (4)))
Input: (((FR 1) 2 / 5) ((FR 2) 2 / 5))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E3: ((*OUTPUT (5)))
Input: (((FR 1) 2 / 3) ((FR 2) 2 / 3))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E3: ((*OUTPUT (3)))
Input: (((FR 1) 1 / 5) ((FR 2) 1 / 5))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E3: ((*OUTPUT (5)))
Input: (((FR 1) 1 / 4) ((FR 2) 1 / 4))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E3: ((*OUTPUT (4)))
Input: (((FR 1) 1 / 3) ((FR 2) 1 / 3))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E3: ((*OUTPUT (3)))
Input: (((FR 1) 1 / 2) ((FR 2) 1 / 2))
CORRECT-E1&2&3: ((*OUTPUT (0)))
ERROR-E3: ((*OUTPUT (2)))
The prediction matches the empirical data.
Number of Possible Problems: 25.
Number of Empirically-derived CPs: 9.
Number of PG-predicted CPs: 9.

PG predictions for models: CORRECT-E4&5 vs. ERROR-E4...
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 4 / 5) ((FR 2) 4 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (5 AND 0 / 5)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 4 / 5) ((FR 2) 4 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (4 AND 0 / 5)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 4 / 5) ((FR 2) 4 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (3 AND 0 / 5)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 4 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (2 AND 0 / 5)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 4 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (1 AND 0 / 5)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (5 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (4 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (3 AND 1 / 5)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 3 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (5 AND 0 / 5)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 3 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (4 AND 0 / 5)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (3 AND 0 / 5)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (2 AND 0 / 5)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (1 AND 0 / 5)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 3 / 4) ((FR 2) 3 / 4))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (5 AND 0 / 4)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 3 / 4) ((FR 2) 3 / 4))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (4 AND 0 / 4)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 3 / 4) ((FR 2) 3 / 4))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (3 AND 0 / 4)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 3 / 4) ((FR 2) 3 / 4))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (2 AND 0 / 4)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 3 / 4) ((FR 2) 3 / 4))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (1 AND 0 / 4)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (2 / 5)))
ERROR-E4: ((*OUTPUT (5 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (2 / 5)))
ERROR-E4: ((*OUTPUT (4 AND 2 / 5)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (2 / 5)))
ERROR-E4: ((*OUTPUT (3 AND 2 / 5)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (2 / 5)))
ERROR-E4: ((*OUTPUT (2 AND 2 / 5)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (2 / 5)))
ERROR-E4: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (5 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (4 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (3 AND 1 / 5)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 2 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (5 AND 0 / 5)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 2 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (4 AND 0 / 5)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (3 AND 0 / 5)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (2 AND 0 / 5)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (1 AND 0 / 5)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 2 / 3) ((FR 2) 2 / 3))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (5 AND 0 / 3)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 2 / 3) ((FR 2) 2 / 3))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (4 AND 0 / 3)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 2 / 3) ((FR 2) 2 / 3))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (3 AND 0 / 3)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 2 / 3) ((FR 2) 2 / 3))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (2 AND 0 / 3)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 2 / 3) ((FR 2) 2 / 3))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (1 AND 0 / 3)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 4 / 5) ((FR 2) 1 / 5))

CORRECT-E4&5: ((*OUTPUT (3 / 5)))
ERROR-E4: ((*OUTPUT (5 AND 3 / 5)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (3 / 5)))
ERROR-E4: ((*OUTPUT (4 AND 3 / 5)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (3 / 5)))
ERROR-E4: ((*OUTPUT (3 AND 3 / 5)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (3 / 5)))
ERROR-E4: ((*OUTPUT (2 AND 3 / 5)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (3 / 5)))
ERROR-E4: ((*OUTPUT (1 AND 3 / 5)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 / 5)))
ERROR-E4: ((*OUTPUT (5 AND 2 / 5)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 / 5)))
ERROR-E4: ((*OUTPUT (4 AND 2 / 5)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 / 5)))
ERROR-E4: ((*OUTPUT (3 AND 2 / 5)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 / 5)))
ERROR-E4: ((*OUTPUT (2 AND 2 / 5)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 / 5)))
ERROR-E4: ((*OUTPUT (1 AND 2 / 5)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (5 AND 1 / 5)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (4 AND 1 / 5)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (3 AND 1 / 5)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (2 AND 1 / 5)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 / 5)))
ERROR-E4: ((*OUTPUT (1 AND 1 / 5)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 1 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (5 AND 0 / 5)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 1 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (4 AND 0 / 5)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 1 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (3 AND 0 / 5)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 1 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (2 AND 0 / 5)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 1 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (1 AND 0 / 5)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (2 / 4)))
ERROR-E4: ((*OUTPUT (5 AND 2 / 4)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (2 / 4)))
ERROR-E4: ((*OUTPUT (4 AND 2 / 4)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (2 / 4)))
ERROR-E4: ((*OUTPUT (3 AND 2 / 4)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (2 / 4)))
ERROR-E4: ((*OUTPUT (2 AND 2 / 4)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (2 / 4)))
ERROR-E4: ((*OUTPUT (1 AND 2 / 4)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 1 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (5 AND 0 / 4)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 1 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (4 AND 0 / 4)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 1 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (3 AND 0 / 4)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 1 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (2 AND 0 / 4)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 1 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (1 AND 0 / 4)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (1 / 3)))
ERROR-E4: ((*OUTPUT (5 AND 1 / 3)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (1 / 3)))
ERROR-E4: ((*OUTPUT (4 AND 1 / 3)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (1 / 3)))
ERROR-E4: ((*OUTPUT (3 AND 1 / 3)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (1 / 3)))
ERROR-E4: ((*OUTPUT (2 AND 1 / 3)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (1 / 3)))
ERROR-E4: ((*OUTPUT (1 AND 1 / 3)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 1 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (5 AND 0 / 3)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 1 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (4 AND 0 / 3)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 1 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (3 AND 0 / 3)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 1 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (2 AND 0 / 3)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 1 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (1 AND 0 / 3)))
Input: (((WN 1) 5) ((WN 2) 5) ((FR 1) 1 / 2) ((FR 2) 1 / 2))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (5 AND 0 / 2)))
Input: (((WN 1) 4) ((WN 2) 4) ((FR 1) 1 / 2) ((FR 2) 1 / 2))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (4 AND 0 / 2)))
Input: (((WN 1) 3) ((WN 2) 3) ((FR 1) 1 / 2) ((FR 2) 1 / 2))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (3 AND 0 / 2)))
Input: (((WN 1) 2) ((WN 2) 2) ((FR 1) 1 / 2) ((FR 2) 1 / 2))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (2 AND 0 / 2)))
Input: (((WN 1) 1) ((WN 2) 1) ((FR 1) 1 / 2) ((FR 2) 1 / 2))
CORRECT-E4&5: ((*OUTPUT (0)))
ERROR-E4: ((*OUTPUT (1 AND 0 / 2)))

The prediction matches the empirical data.
Number of Possible Problems: 125.
Number of Empirically-derived CPs: 85.
Number of PG-predicted CPs: 85.

PG predictions for models: CORRECT-E4&5 vs. ERROR-E5...

Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (3 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (4 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (3 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 3 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (3 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (4 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (3 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))

Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (3 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (4 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (3 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 2 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-E5: ((*OUTPUT (3 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-E5: ((*OUTPUT (3 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-E5: ((*OUTPUT (3 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (3 AND 3 / 5)))
ERROR-E5: ((*OUTPUT (3 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-E5: ((*OUTPUT (3 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-E5: ((*OUTPUT (3 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (4 AND 3 / 5)))
ERROR-E5: ((*OUTPUT (3 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (3 AND 3 / 5)))
ERROR-E5: ((*OUTPUT (3 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 3 / 5)))
ERROR-E5: ((*OUTPUT (3 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 4 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 3 / 5)))
ERROR-E5: ((*OUTPUT (3 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (3 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (4 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (3 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 3 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 2 / 5)))
ERROR-E5: ((*OUTPUT (2 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (3 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (4 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (3 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (2 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 2 / 5) ((FR 2) 1 / 5))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 5)))
ERROR-E5: ((*OUTPUT (1 / 5)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (1 AND 2 / 4)))
ERROR-E5: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (2 AND 2 / 4)))
ERROR-E5: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (1 AND 2 / 4)))
ERROR-E5: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (3 AND 2 / 4)))
ERROR-E5: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (2 AND 2 / 4)))
ERROR-E5: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (1 AND 2 / 4)))
ERROR-E5: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (4 AND 2 / 4)))
ERROR-E5: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (3 AND 2 / 4)))
ERROR-E5: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (2 AND 2 / 4)))
ERROR-E5: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 3 / 4) ((FR 2) 1 / 4))
CORRECT-E4&5: ((*OUTPUT (1 AND 2 / 4)))
ERROR-E5: ((*OUTPUT (2 / 4)))
Input: (((WN 1) 5) ((WN 2) 4) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 3)))
ERROR-E5: ((*OUTPUT (1 / 3)))
Input: (((WN 1) 5) ((WN 2) 3) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (2 AND 1 / 3)))
ERROR-E5: ((*OUTPUT (1 / 3)))
Input: (((WN 1) 4) ((WN 2) 3) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 3)))
ERROR-E5: ((*OUTPUT (1 / 3)))
Input: (((WN 1) 5) ((WN 2) 2) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (3 AND 1 / 3)))
ERROR-E5: ((*OUTPUT (1 / 3)))
Input: (((WN 1) 4) ((WN 2) 2) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (2 AND 1 / 3)))
ERROR-E5: ((*OUTPUT (1 / 3)))
Input: (((WN 1) 3) ((WN 2) 2) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 3)))
ERROR-E5: ((*OUTPUT (1 / 3)))
Input: (((WN 1) 5) ((WN 2) 1) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (4 AND 1 / 3)))
ERROR-E5: ((*OUTPUT (1 / 3)))
Input: (((WN 1) 4) ((WN 2) 1) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (3 AND 1 / 3)))
ERROR-E5: ((*OUTPUT (1 / 3)))
Input: (((WN 1) 3) ((WN 2) 1) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (2 AND 1 / 3)))
ERROR-E5: ((*OUTPUT (1 / 3)))
Input: (((WN 1) 2) ((WN 2) 1) ((FR 1) 2 / 3) ((FR 2) 1 / 3))
CORRECT-E4&5: ((*OUTPUT (1 AND 1 / 3)))
ERROR-E5: ((*OUTPUT (1 / 3)))
The prediction matches the empirical data.
Number of Possible Problems: 250.
Number of Empirically-derived CPs: 80.
Number of PG-predicted CPs: 80.

Appendix IVe

Full Trace of PG's Analysis of Model H1

This is the full trace of PG's analysis of the difference between models CORRECT-H1 and ERROR-H1. Comments are in italics.

This is the Lisp form which invokes the run. The axioms and cached clauses are reinitialised (i.e. initialised to the empty set). PG is then provided with the three equality axioms, and one pertaining to the 'less than' predicate (LT). The search is then invoked via a call to Gimme-CP, which takes the two models, a set of Input Specifications, a start flag and a comment.

```
> (progn (reinitialise-axioms)
    (reinitialise-known-clause-set)
    (AX ∀(x) EQ(x,x)) ;Reflexivity
    (AX ∀(x,y) EQ(x,y) ⊃ EQ(y,x)) ;Symmetry
    (AX ∀(x,y,z) EQ(x,y) ∧ EQ(y,z) ⊃ EQ(x,z)) ;Transitivity
    (AX ∀(x,y) EQ(x,y) ⊃ ¬LT(x,y))
    (Gimme-CP 'CORRECT-h1 'ERROR-h1
          '((((fr 1) ?nx / ?d) ((fr 2) ?ny / ?d) (< ?nx ?d) (< ?ny ?d)))
          T
          "do not cancel."))
```

CP-search for rulesets CORRECT-H1 and ERROR-H1,
on Input Specs: ((((FR 1) ?NX / ?D) ((FR 2) ?NY / ?D) (< ?NX ?D) (< ?NY ?D)))
In the error model, do not cancel.

In the following trace, 'CS' denotes the Conflict Set, and the numbers 1, 2, and 3, on the same line, denote the application of Refractoriness, Recency, and Specificity, respectively.

Choosing Input Specification: (((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0) (< ?NX.0 ?D.0) (< ?NY.0 ?D.0))

[1]
Context: CORRECT-H1
Trying: NM2A... Trying: NM2B... Trying: DN1... Trying: HCF... Trying: END1...
Trying: HLT...
CS: (#DN1), 1: (#DN1), 2: (#DN1), 3: (#DN1),
#DN1 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

[2]
Context: ERROR-H1
Trying: NM2A... Trying: NM2B... Trying: DN1... Trying: END1... Trying: HLT...
CS: (#DN1), 1: (#DN1), 2: (#DN1), 3: (#DN1),
#DN1 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: CORRECT-H1/#DN1 with ERROR-H1/#DN1 is consistent.
(Initialisation Complete)

Having completed the initialisation, PG selects the model CORRECT-H1 and fires the instantiation #DN1 which deposits the denominator, ?d.0, in Working Memory.

Selecting context: CORRECT-H1

[1]Firing: #DN1
Deposited: ((/ ?D.0))
WM: ((/ ?D.0) ((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0))
Constraints: (LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Exclusions: NIL

PG now considers the rules NM2A and NM2B which calculate the numerator.

[3]
Context: CORRECT-H1
Trying: NM2A...
Checking pattern-set consistency for #NM2A...

Satisfiables: (LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: LT(?ny.0,?nx.0)... is consistent.

PG fails to prove that ?ny.0 is less than ?nx.0 (by trying to refute its negation). Nevertheless it could be true; PG asks the user to confirm that this is in fact correct. Failure to falsify the negation of a theorem means that the theorem cannot be deduced from the current set of axioms. In this trace, any item in double angled brackets is a request from PG to the user. Thus, the expression: ***<<Seems consistent to me. Is it?>>*** *is a request; it is immediately followed by the users response (in this case "Yes").*
<<Seems consistent to me. Is it?>> Yes. Trying: NM2B...
Checking pattern-set consistency for #NM2B...
Satisfiables: (LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: EQ(?nx.0,?ny.0)

Here PG asks for more time to prove the theorem (in all runs, PG is only allowed 10 seconds to find a refutation). By entering the value '0', the user signifies that PG can spend no more time on this problem.
<<How many more seconds can I have?>>0

Because it failed to find a refutation in the allotted 10 seconds, PG asks the user to perform the proof. The user tells PG that the negated theorem is in fact consistent.
<<Consistent?>> (EQ(?nx.0,?ny.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0)) ...Yes ... is consistent.
Satisfiables: (LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: EQ(?nx.0,?ny.0) ... is consistent.
Trying: DN1...
MG-Neg-Nullifier: #DN1, (NIL)
Satisfiables: (LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))

The empty clause, NIL, is always false.
Negated Theorem: (NIL)... is not consistent.
#DN1 was rejected by the negation filter.Trying: HCF... Trying: END1...
Trying: HLT...

#NM2A and #NM2B are equally valid instantiations, i.e. conflict resolution cannot choose between them. Therefore, PG builds an exclusion clause for each one. #NM2A would vanquish #NM2B in cases where ¬EQ(?nx.0,?ny.0), because #NM2B requires that ?nx and ?ny be equal. Similary, #NM2A requires that ?ny be less than ?nx, therefore #NM2B is guaranteed to fire when ¬LT(?ny.0,?nx.0).
CS: (#NM2A #NM2B), 1: (#NM2A #NM2B), 2: (#NM2B #NM2A), 3: (#NM2A #NM2B),
#NM2A
has the following competitors: (#NM2B), yielding exclusion clauses: ¬EQ(?nx.0,?ny.0)
#NM2B
has the following competitors: (#NM2A), yielding exclusion clauses: ¬LT(?ny.0,?nx.0)
Checking exclusion clauses for #NM2A...
Satisfiables: (LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: ¬EQ(?nx.0,?ny.0)... is consistent.
<<Seems consistent to me. Is it?>> Yes.
Checking exclusion clauses for #NM2B...
Satisfiables: (LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: (EQ(?nx.0,?ny.0) ∧ ¬LT(?ny.0,?ny.0))
<<How many more seconds can I have?>>0
<<Consistent?>> (EQ(?nx.0,?ny.0) ∧ ¬LT(?ny.0,?ny.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0)) ...Yes... is consistent.
Contradictory instantiations: NIL

The pairing: CORRECT-H1/#NM2A with ERROR-H1/#DN1 is consistent.
Checking consistency of State Pair...
Satisfiables: (EQ(?nx.0,?ny.0) ∧ LT(?ny.0,?d.0) ∧ ¬LT(?ny.0,?ny.0))
Negated Theorem: LT(?nx.0,?d.0) ... is consistent.
The pairing: CORRECT-H1/#NM2B with ERROR-H1/#DN1 is consistent.

PG now selects the path emanating from #NM2A. It will return to the other path in frame [3] at the very end of this trace (#NM2B) when the current path has been fully explored.
[3]Firing: #NM2A
Deposited: (((*SUBTRACT ?NX.0 ?NY.0) /))
WM: (((*SUBTRACT ?NX.0 ?NY.0) /) (/ ?D.0) ((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0))
Constraints: (LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Exclusions: ¬EQ(?nx.0,?ny.0)

[4]_____
Context: CORRECT-H1
Trying: NM2A... Trying: NM2B...
Checking pattern-set consistency for #NM2B...
Satisfiables: (LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: EQ(?nx.0,?ny.0)... is not consistent.Trying: DN1...
MG-Neg-Nullifier: #DN1, (NIL)
Satisfiables: (LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: (NIL) ... is not consistent.
#DN1 was rejected by the negation filter.Trying: HCF...
Checking pattern-set consistency for #HCF...
Satisfiables: (LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))

Negated Theorem: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0)
∧ LT(SUB(?nx.0,?ny.0),?d.0))... is consistent.
<<Seems consistent to me. Is it?>> Yes. Trying: END1...
Checking pattern-set consistency for #END1...
Satisfiables: (LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: ¬EQ(SUB(?nx.0,?ny.0),0) ... is consistent.
Trying: HLT...
CS: (#NM2A #HCF #END1), 1: (#HCF #END1), 2: (#END1 #HCF), 3: (#HCF),

#HCF wins over #END1 because it is more specific.
#HCF has the following competitors: NIL, yielding exclusion clauses: NIL

Contradictory instantiations: NIL
2: (#END1), 3: (#END1),

However, #END1 could fire if #HCF's constraints are violated as in the exclusion clause below.
#END1
has the following competitors: (#HCF), yielding exclusion clauses: (EQ(SUB(?nx.0, ?ny.0),0)
∨ ¬NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∨ ¬LT(SUB(?nx.0,?ny.0),?d.0))
Checking exclusion clauses for #END1...

Satisfiables: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0)
∧ LT(?ny.0,?d.0) ∧ ¬EQ(?nx.0,?ny.0))

Negated Theorem: (EQ(SUB(?nx.0,?ny.0),0) ∨ ¬NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0)
∨ ¬LT(SUB(?nx.0,?ny.0),?d.0))... is consistent.
<<Seems consistent to me. Is it?>> Yes.
Contradictory instantiations: NIL

The pairing: CORRECT-H1/#HCF with ERROR-H1/#DN1 is consistent.
The pairing: CORRECT-H1/#END1 with ERROR-H1/#DN1 is consistent.

#HCF fires and removes the cancellable numerator and denominator from Working Memory. These are replaced by a new numerator and denominator where a cancel operation has been performed.
[4]Firing: #HCF
Deleting: ((*SUBTRACT ?NX.0 ?NY.0) /)
Deleting: (/ ?D.0)
Deposited: (((*CANCEL-NM (*SUBTRACT ?NX.0 ?NY.0) ?D.0) /) (/ (*CANCEL-DN (*SUBTRACT ?NX.0 ?NY.0) ?D.0))
(HAVE CANCELLED))
WM: ((HAVE CANCELLED) (/ (*CANCEL-DN (*SUBTRACT ?NX.0 ?NY.0) ?D.0))
((*CANCEL-NM (*SUBTRACT ?NX.0 ?NY.0) ?D.0) /) ((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0))

Constraints: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0)
∧ LT(SUB(?nx.0,?ny.0),?d.0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Exclusions: ¬EQ(?nx.0,?ny.0)

[5]_____
Context: CORRECT-H1

PG again matches all of the rules against Working Memory. Much of this is a waste of time because the instantiations will be discarded, having fired on earlier cycles.
Trying: NM2A...
Checking pattern-set consistency for #NM2A...

Satisfiables: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0)
∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: EQ(?d.0,CANCEL-DN(SUB(?nx.0,?ny.0),?d.0))
<<How many more seconds can I have?>>0

<<Consistent?>> (EQ(?d.0,CANCEL-DN(SUB(?nx.0,?ny.0),?d.0)) ∧ ¬EQ(SUB(?nx.0,?ny.0),0)
∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0)
∧ LT(?ny.0,?d.0)) ...No... is not consistent.
Trying: NM2B...
Checking pattern-set consistency for #NM2B...

Satisfiables: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0)
∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: (EQ(?nx.0,?ny.0) ∧ EQ(?d.0,CANCEL-DN(SUB(?ny.0,?ny.0),?d.0)))
... is not consistent.Trying: DN1...
MG-Neg-Nullifier: #DN1, (NIL)

Satisfiables: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0)
∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: (NIL) ... is not consistent.
#DN1 was rejected by the negation filter.Trying: HCF...
Checking pattern-set consistency for #HCF...

Satisfiables: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0)
∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))

Negated Theorem: (¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0)
∧ NEEDS-CANCELLING-P(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),CANCEL-DN(SUB(?nx.0,?ny.0),?d.0))
∧ LT(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),CANCEL-DN(SUB(?nx.0,?ny.0),?d.0)))... is consistent.
<<Seems consistent to me. Is it?>> No. Trying: END1...
Checking pattern-set consistency for #END1...

Satisfiables: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0)
∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: ¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0)... is consistent.
<<Seems consistent to me. Is it?>> Yes. Trying: HLT...
CS: (#END1), 1: (#END1), 2: (#END1), 3: (#END1),
#END1 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: CORRECT-H1/#END1 with ERROR-H1/#DN1 is consistent.

The instantiation #END1 fires and deposits the cancelled answer in Working Memory.
[5]Firing: #END1
Deposited: ((ANSWER IS ((*CANCEL-NM (*SUBTRACT ?NX.0 ?NY.0) ?D.0) /
(*CANCEL-DN (*SUBTRACT ?NX.0 ?NY.0) ?D.0))))

WM: ((ANSWER IS ((*CANCEL-NM (*SUBTRACT ?NX.0 ?NY.0) ?D.0) /
(*CANCEL-DN (*SUBTRACT ?NX.0 ?NY.0) ?D.0))) (HAVE CANCELLED)
(/ (*CANCEL-DN (*SUBTRACT ?NX.0 ?NY.0) ?D.0))
((*CANCEL-NM (*SUBTRACT ?NX.0 ?NY.0) ?D.0) /) ((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0))

Constraints: (¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0) ∧ ¬EQ(SUB(?nx.0,?ny.0),0)
∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0)
∧ LT(?ny.0,?d.0))
Exclusions: ¬EQ(?nx.0,?ny.0)

[6]_____
Context: CORRECT-H1

Most of the pattern matching and theorem proving, below, happened on the previous cycle. PG is very inefficient because it repeats everything on every cycle.
Trying: NM2A...
Checking pattern-set consistency for #NM2A...

Satisfiables: (¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0) ∧ ¬EQ(SUB(?nx.0,?ny.0),0)
∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0)
∧ LT(?ny.0,?d.0))
Negated Theorem: EQ(?d.0,CANCEL-DN(SUB(?nx.0,?ny.0),?d.0))
<<How many more seconds can I have?>>0

<<Consistent?>> (EQ(?d.0,CANCEL-DN(SUB(?nx.0,?ny.0),?d.0)) ∧ ¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0)
∧ ¬EQ(SUB(?nx.0,?ny.0),0) ∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0)
∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0)) ...No... is not consistent.Trying: NM2B...
Checking pattern-set consistency for #NM2B...

Satisfiables: (¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0) ∧ ¬EQ(SUB(?nx.0,?ny.0),0)
∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0)
∧ LT(?ny.0,?d.0))
Negated Theorem: (EQ(?nx.0,?ny.0) ∧ EQ(?d.0,CANCEL-DN(SUB(?ny.0,?ny.0),?d.0)))
... is not consistent.Trying: DN1...
MG-Neg-Nullifier: #DN1, (NIL)

Satisfiables: (¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0) ∧ ¬EQ(SUB(?nx.0,?ny.0),0)
∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0)
∧ LT(?ny.0,?d.0))
Negated Theorem: (NIL) ... is not consistent.
#DN1 was rejected by the negation filter.Trying: HCF...
Checking pattern-set consistency for #HCF...

Satisfiables: (¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0) ∧ ¬EQ(SUB(?nx.0,?ny.0),0)
∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0)
∧ LT(?ny.0,?d.0))

Negated Theorem: (NEEDS-CANCELLING-P(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),CANCEL-DN(SUB(?nx.0,?ny.0),?d.0))
∧ LT(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),CANCEL-DN(SUB(?nx.0,?ny.0),?d.0)))... is consistent.
<<Seems consistent to me. Is it?>> No. Trying: END1... Trying: HLT...
CS: (#END1 #HLT), 1: (#HLT), 2: (#HLT), 3: (#HLT),
#HLT has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: CORRECT-H1/#HLT with ERROR-H1/#DN1 is consistent.

Having computed the answer, PG now halts and lists the constraints and outputs produced by analysing this path.
[6]Firing: #HLT
WM: ((ANSWER IS ((*CANCEL-NM (*SUBTRACT ?NX.0 ?NY.0) ?D.0) /
(*CANCEL-DN (*SUBTRACT ?NX.0 ?NY.0) ?D.0))) (HAVE CANCELLED)
(/ (*CANCEL-DN (*SUBTRACT ?NX.0 ?NY.0) ?D.0))
((*CANCEL-NM (*SUBTRACT ?NX.0 ?NY.0) ?D.0) /) ((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0))

Constraints: (¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0) ∧ ¬EQ(SUB(?nx.0,?ny.0),0)
∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0)
∧ LT(?ny.0,?d.0))
Exclusions: ¬EQ(?nx.0,?ny.0)
New outputs: ((*OUTPUT ((*CANCEL-NM (*SUBTRACT ?NX.0 ?NY.0) ?D.0) /
(*CANCEL-DN (*SUBTRACT ?NX.0 ?NY.0) ?D.0))))
Halt signalled on this path.

Control now switches to the error model. PG could carry on analysing CORRECT-H1, however, if one were trying to find a CP as soon as possible, then now would be the time to switch. PG does so.
Switching rulesets because of *HALT.

_____
Selecting context: ERROR-H1

This model follows a similar path to that of CORRECT-H1 but does not cancel.

[2]Firing: #DN1
Deposited: ((/ ?D.0))
WM: ((/ ?D.0) ((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0))
Constraints: (LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Exclusions: NIL

[7]_____
Context: ERROR-H1

Trying: NM2A...
Checking pattern-set consistency for #NM2A...
  Satisfiables: (LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
  Negated Theorem: LT(?ny.0,?nx.0) ... is consistent.
Trying: NM2B...
Checking pattern-set consistency for #NM2B...
  Satisfiables: (LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
  Negated Theorem: EQ(?nx.0,?ny.0) ... is consistent.
  Satisfiables: (LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
  Negated Theorem: EQ(?nx.0,?ny.0) ... is consistent.
Trying: DN1...
MG-Neg-Nullifier: #DN1, (NIL)
  Satisfiables: (LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
  Negated Theorem: (NIL) ... is not consistent.
#DN1 was rejected by the negation filter.Trying: END1... Trying: HLT...
CS: (#NM2A #NM2B), 1: (#NM2A #NM2B), 2: (#NM2B #NM2A), 3: (#NM2A #NM2B),
#NM2A
has the following competitors: (#NM2B), yielding exclusion clauses: ¬EQ(?nx.0,?ny.0)
#NM2B
has the following competitors: (#NM2A), yielding exclusion clauses: ¬LT(?ny.0,?nx.0)
Checking exclusion clauses for #NM2A...
  Satisfiables: (LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
  Negated Theorem: ¬EQ(?nx.0,?ny.0) ... is consistent.
Checking exclusion clauses for #NM2B...
  Satisfiables: (LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
  Negated Theorem: (EQ(?nx.0,?ny.0) ∧ ¬LT(?ny.0,?ny.0))
... is consistent.
Contradictory instantiations: NIL

The pairing: CORRECT-H1/#HLT with ERROR-H1/#NM2A is consistent.
Checking consistency of State Pair...
 Satisfiables: (¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0) ∧ ¬EQ(SUB(?nx.0,?ny.0),0)
 ∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0) ∧ LT(?ny.0,?nx.0)
 ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0) ∧ ¬EQ(?nx.0,?ny.0))
  Negated Theorem: (EQ(?nx.0,?ny.0) ∧ ¬LT(?ny.0,?ny.0))
... is not consistent.
The pairing: CORRECT-H1/#HLT with ERROR-H1/#NM2B is inconsistent.
The pairing: CORRECT-H1/#END1 with ERROR-H1/#NM2A is consistent.
Checking consistency of State Pair...

 Satisfiables: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0)
 ∧ ¬LT(SUB(?nx.0,?ny.0),?d.0) ∧ ¬NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0)
 ∧ EQ(SUB(?nx.0,?ny.0),0) ∧ ¬EQ(?nx.0,?ny.0))
  Negated Theorem: (EQ(?nx.0,?ny.0) ∧ ¬LT(?ny.0,?ny.0))
... is not consistent.
The pairing: CORRECT-H1/#END1 with ERROR-H1/#NM2B is inconsistent.
Checking consistency of State Pair...
  Satisfiables: (EQ(?nx.0,?ny.0) ∧ LT(?ny.0,?d.0) ∧ ¬LT(?ny.0,?ny.0))
  Negated Theorem: (LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ ¬EQ(?nx.0,?ny.0))
... is not consistent.
The pairing: CORRECT-H1/#NM2B with ERROR-H1/#NM2A is inconsistent.
The pairing: CORRECT-H1/#NM2B with ERROR-H1/#NM2B is consistent.

[7]Firing: #NM2A
Deposited: (((*SUBTRACT ?NX.0 ?NY.0) /))
WM: (((*SUBTRACT ?NX.0 ?NY.0) /) (/ ?D.0) ((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0))
Constraints: (LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Exclusions: ¬EQ(?nx.0,?ny.0)

[8]_____
Context: ERROR-H1
Trying: NM2A... Trying: NM2B...
Checking pattern-set consistency for #NM2B...
  Satisfiables: (LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
  Negated Theorem: EQ(?nx.0,?ny.0) ... is not consistent.
Trying: DN1...
MG-Neg-Nullifier: #DN1, (NIL)
  Satisfiables: (LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
  Negated Theorem: (NIL) ... is not consistent.
#DN1 was rejected by the negation filter.Trying: END1...
Checking pattern-set consistency for #END1...
  Satisfiables: (LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
  Negated Theorem: ¬EQ(SUB(?nx.0,?ny.0),0) ... is consistent.
Trying: HLT...
CS: (#NM2A #END1), 1: (#END1), 2: (#END1), 3: (#END1),
#END1 has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: CORRECT-H1/#HLT with ERROR-H1/#END1 is consistent.
The pairing: CORRECT-H1/#END1 with ERROR-H1/#END1 is consistent.

The answer is generated, but this error model does not try to cancel.
[8]Firing: #END1
Deposited: ((ANSWER IS ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)))
WM: ((ANSWER IS ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)) ((*SUBTRACT ?NX.0 ?NY.0) /)
(/ ?D.0) ((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0))

Constraints: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))

Exclusions: ¬EQ(?nx.0,?ny.0)

[9]_____
Context: ERROR-H1
Trying: NM2A... Trying: NM2B...
Checking pattern-set consistency for #NM2B...

Satisfiables: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: EQ(?nx.0,?ny.0) ... is not consistent.
Trying: DN1...
MG-Neg-Nullifier: #DN1, (NIL)

Satisfiables: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: (NIL) ... is not consistent.
#DN1 was rejected by the negation filter.Trying: END1... Trying: HLT...
CS: (#NM2A #END1 #HLT), 1: (#HLT), 2: (#HLT), 3: (#HLT),
#HLT has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL

The pairing: CORRECT-H1/#HLT with ERROR-H1/#HLT is consistent.
The pairing: CORRECT-H1/#END1 with ERROR-H1/#HLT is consistent.

[9]Firing: #HLT
WM: ((ANSWER IS ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)) ((*SUBTRACT ?NX.0 ?NY.0) /) (/ ?D.0)
((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0))

Constraints: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Exclusions: ¬EQ(?nx.0,?ny.0)
New outputs: ((*OUTPUT ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)))
Halt signalled on this path.

Both models have generated some output, so it is now time to compare them and see if the input/output mappings constitute a CP.

CP-generation info:
Input Spec: (((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0) (< ?NX.0 ?D.0) (< ?NY.0 ?D.0))

State information for STATE1. Ruleset: CORRECT-H1
Instantiations: (#DN1 #NM2A #HCF #END1 #HLT)
Outputs: ((*OUTPUT ((*CANCEL-NM (*SUBTRACT ?NX.0 ?NY.0) ?D.0) /
(*CANCEL-DN (*SUBTRACT ?NX.0 ?NY.0) ?D.0))))

(¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0) ∧ ¬EQ(SUB(?nx.0,?ny.0),0)
∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0) ∧ LT(?ny.0,?nx.0)
∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Exclusion clauses: ¬EQ(?nx.0,?ny.0)
This ruleset is in a halt state.

State information for STATE2. Ruleset: ERROR-H1
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)))
(¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Exclusion clauses: ¬EQ(?nx.0,?ny.0)
This ruleset is in a halt state.

Satisfiables: (¬EQ(?nx.0,?ny.0) ∧ ¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0)
∧ ¬EQ(SUB(?nx.0,?ny.0),0) ∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0)
∧ LT(SUB(?nx.0,?ny.0),?d.0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))

Negated Theorem: (¬EQ(?d.0,CANCEL-DN(SUB(?nx.0,?ny.0),?d.0))
∨ ¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),SUB(?nx.0,?ny.0)))
<<How many more seconds can I have?>>0

<<Consistent?>> ((¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),SUB(?nx.0,?ny.0))
∨ ¬EQ(?d.0,CANCEL-DN(SUB(?nx.0,?ny.0),?d.0))) ∧ ¬EQ(?nx.0,?ny.0)
∧ ¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0) ∧ ¬EQ(SUB(?nx.0,?ny.0),0)
∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0) ∧ LT(?ny.0,?nx.0)
∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0)) ...Yes... is consistent.

The outputs ((*CANCEL-NM (*SUBTRACT ?NX.0 ?NY.0) ?D.0) /
(*CANCEL-DN (*SUBTRACT ?NX.0 ?NY.0) ?D.0)) and ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)
can be unified. Therefore, a CP could be generated provided the following constraint-set is instantiable:
((¬EQ(?d.0,CANCEL-DN(SUB(?nx.0,?ny.0),?d.0))
∨ ¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),SUB(?nx.0,?ny.0)))
∧ ¬EQ(?nx.0,?ny.0)
∧ ¬EQ(CANCEL-NM(SUB(?nx.0,?ny.0),?d.0),0)
∧ ¬EQ(SUB(?nx.0,?ny.0),0)
∧ NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0)
∧ LT(SUB(?nx.0,?ny.0),?d.0)
∧ LT(?ny.0,?nx.0)
∧ LT(?nx.0,?d.0)
∧ LT(?ny.0,?d.0))

The above constraint expression looks quite complex, however, the key expression is the fifth conjunct. Basically, it means that the CP should be one where the act of subtracting the two denominators leads to a term which requires cancelling.
Switching rulesets because of *HALT.

PG now returns to the model CORRECT-H1 and takes up the path where the answer does not require cancelling.
Selecting context: CORRECT-H1

[4]Firing: #END1
Deposited: ((ANSWER IS ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)))
WM: ((ANSWER IS ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)) ((*SUBTRACT ?NX.0 ?NY.0) /) (/ ?D.0)
((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0))

Constraints: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))

Exclusions: ((EQ(SUB(?nx.0,?ny.0),0) ∨ ¬NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0)
∨ ¬LT(SUB(?nx.0,?ny.0),?d.0)) ∧ ¬EQ(?nx.0,?ny.0))

[10]_____
Context: CORRECT-H1
Trying: NM2A... Trying: NM2B...
Checking pattern-set consistency for #NM2B...

Satisfiables: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: EQ(?nx.0,?ny.0) ... is not consistent.
Trying: DN1...
MG-Neg-Nullifier: #DN1, (NIL)

Satisfiables: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Negated Theorem: (NIL) ... is not consistent.
#DN1 was rejected by the negation filter.Trying: HCF...
Checking pattern-set consistency for #HCF...

Satisfiables: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))

Negated Theorem: (NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0)
∧ LT(SUB(?nx.0,?ny.0),?d.0)) ... is consistent.Trying: END1... Trying: HLT...
CS: (#NM2A #HCF #END1 #HLT), 1: (#HCF #HLT), 2: (#HLT), 3: (#HLT),
#HLT has the following competitors: NIL, yielding exclusion clauses: NIL
Contradictory instantiations: NIL
2: (#HCF), 3: (#HCF),
#HCF has the following competitors: (#HLT), yielding exclusion clauses: (NIL)
Checking exclusion clauses for #HCF...

Satisfiables: (NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0) ∧ LT(SUB(?nx.0,?ny.0),?d.0)
∧ ¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0)
∧ (EQ(SUB(?nx.0,?ny.0),0) ∨ ¬NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0)
∨ ¬LT(SUB(?nx.0,?ny.0),?d.0)) ∧ ¬EQ(?nx.0,?ny.0))
Negated Theorem: (NIL) ... is not consistent.
Contradictory instantiations: (#HCF)

The pairing: CORRECT-H1/#HLT with ERROR-H1/#HLT is consistent.

[10]Firing: #HLT
WM: ((ANSWER IS ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)) ((*SUBTRACT ?NX.0 ?NY.0) /) (/ ?D.0)
((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0))

Constraints: (¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))

Exclusions: ((EQ(SUB(?nx.0,?ny.0),0) ∨ ¬NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0)
∨ ¬LT(SUB(?nx.0,?ny.0),?d.0)) ∧ ¬EQ(?nx.0,?ny.0))
New outputs: ((*OUTPUT ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)))
Halt signalled on this path.

CP-generation info:
Input Spec: (((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0) (< ?NX.0 ?D.0) (< ?NY.0 ?D.0))

State information for STATE1. Ruleset: CORRECT-H1
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)))
(¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))

Exclusion clauses: ((EQ(SUB(?nx.0,?ny.0),0) ∨ ¬NEEDS-CANCELLING-P(SUB(?nx.0,?ny.0),?d.0)
∨ ¬LT(SUB(?nx.0,?ny.0),?d.0)) ∧ ¬EQ(?nx.0,?ny.0))
This ruleset is in a halt state.

State information for STATE2. Ruleset: ERROR-H1
Instantiations: (#DN1 #NM2A #END1 #HLT)
Outputs: ((*OUTPUT ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)))
(¬EQ(SUB(?nx.0,?ny.0),0) ∧ LT(?ny.0,?nx.0) ∧ LT(?nx.0,?d.0) ∧ LT(?ny.0,?d.0))
Exclusion clauses: ¬EQ(?nx.0,?ny.0)
This ruleset is in a halt state.

In the case where the answer does not require cancelling, no CP exists.
No CP can be generated for the outputs: ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0)
and ((*SUBTRACT ?NX.0 ?NY.0) / ?D.0), as they are equal.
Switching rulesets because of *HALT.

PG now switches back to the error model and picks up the NM2B path. This path is followed when the two numerators are equal.
Selecting context: ERROR-H1

[7]Firing: #NM2B
Deposited: ((0 /))
WM: ((0 /) (/ ?D.0) ((FR 1) ?NY.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0))
Constraints: (LT(?ny.0,?d.0) ∧ LT(?ny.0,?d.0))
Exclusions: ¬LT(?ny.0,?ny.0)

[11]_____
Context: ERROR-H1
Trying: NM2A...
Checking pattern-set consistency for #NM2A...
Satisfiables: LT(?ny.0,?d.0)
Negated Theorem: LT(?ny.0,?ny.0)... is not consistent.Trying: NM2B...
Trying: DN1...
MG-Neg-Nullifier: #DN1, (NIL)
Satisfiables: LT(?ny.0,?d.0)
Negated Theorem: (NIL)... is not consistent.
#DN1 was rejected by the negation filter.Trying: END1...

This model has no rule for catering for cases where the answer's numerator is zero. END1 requires that the numerator be positive. Thus, PG discards this path as no further processing is possible.
Checking pattern-set consistency for #END1...
Satisfiables: LT(?ny.0,?d.0)
Negated Theorem: ¬EQ(0,0)... is not consistent.Trying: HLT...
CS: (#NM2B), 1: NIL,
Switching ruleset because the current ruleset has no unhalted State Pairs.

PG now picks up the NM2B path which it suspended near the beginning of this trace.
Selecting context: CORRECT-H1

[3]Firing: #NM2B
Deposited: ((0 /))
WM: ((0 /) (/ ?D.0) ((FR 1) ?NY.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0))
Constraints: (LT(?ny.0,?d.0) ∧ LT(?ny.0,?d.0))
Exclusions: ¬LT(?ny.0,?ny.0)

[12]_____
Context: CORRECT-H1
Trying: NM2A...
Checking pattern-set consistency for #NM2A...
Satisfiables: LT(?ny.0,?d.0)
Negated Theorem: LT(?ny.0,?ny.0) ... is not consistent.
Trying: NM2B... Trying: DN1...
MG-Neg-Nullifier: #DN1, (NIL)
Satisfiables: LT(?ny.0,?d.0)
Negated Theorem: (NIL) ... is not consistent.
#DN1 was rejected by the negation filter.Trying: HCF...
Checking pattern-set consistency for #HCF...
Satisfiables: LT(?ny.0,?d.0)
Negated Theorem: (¬EQ(0,0) ∧ NEEDS-CANCELLING-P(0,?d.0) ∧ LT(0,?d.0))
... is not consistent.Trying: END1...

As with ERROR-H1, END1 cannot be instantiated if the answer's numerator is zero.
Checking pattern-set consistency for #END1...
Satisfiables: LT(?ny.0,?d.0)
Negated Theorem: ¬EQ(0,0)... is not consistent.Trying: HLT...
CS: (#NM2B), 1: NIL,

At this point, both models fail to produce further outputs and so processing ceases.

CP-generation info:
Input Spec: (((FR 1) ?NX.0 / ?D.0) ((FR 2) ?NY.0 / ?D.0) (< ?NX.0 ?D.0) (< ?NY.0 ?D.0))

State information for STATE1. Ruleset: CORRECT-H1
Instantiations: (#DN1 #NM2B)
Outputs: NIL
This ruleset is in a halt state.

State information for STATE2. Ruleset: ERROR-H1
Instantiations: (#DN1 #NM2B)
Outputs: NIL
This ruleset is in a halt state.
Switching ruleset because the current ruleset has no unhalted State Pairs.

Selecting context: ERROR-H1

No input specifications left.